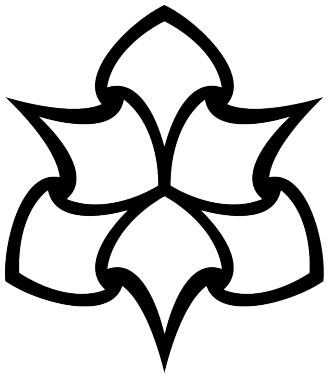


MANCHESTER METROPOLITAN UNIVERSITY

Hardware Acceleration of Network Intrusion Detection System Using FPGA

by

Adeel Hashmi

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Science and Engineering
School of Computing, Mathematics and Digital Technology

January 2011

Acknowledgements

I would like to express my gratitude to my PhD supervisors Dr. Andy Nisbet and Mr. Clive Mingham for the guidance and support. Throughout my thesis writing period, they provided encouragement, sound advice, good teaching, and lots of good ideas. I would have been lost without them.

I am grateful to my office mates for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Pete, Naomi and John for their valuable feedbacks and providing me regular cups of coffee.

Special thanks to my friends Nauman and Kaleem for helping me get through the difficult times, and for all the emotional support, entertainment, and caring they provided.

Lastly, and most importantly, I wish to thank my family for their support, patience and understanding. To them I dedicate this thesis.

Abstract

This thesis presents new algorithms and hardware designs for Signature-based Network Intrusion Detection System (SB-NIDS) optimisation exploiting a hybrid hardware-software co-designed embedded processing platform. The work describe concentrates on optimisation of a complete SB-NIDS *Snort* application software on a FPGA based hardware-software target rather than on the implementation of a single functional unit for hardware acceleration. Pattern Matching Hardware Accelerator (PMHA) based on Bloom filter was designed to optimise SB-NIDS performance for execution on a Xilinx MicroBlaze soft-core processor. The Bloom filter approach enables the potentially large number of network intrusion attack patterns to be efficiently represented and searched primarily using accesses to FPGA on-chip memory. The thesis demonstrates, the viability of hybrid hardware-software co-designed approach for SB-NIDS. Future work is required to investigate the effects of later generation FPGA technology and multi-core processors in order to clearly prove the benefits over conventional processor platforms for SB-NIDS.

The strengths and weaknesses of the hardware accelerators and algorithms are analysed, and experimental results are examined to determine the effectiveness of the implementation. Experimental results confirm that the PMHA is capable of performing network packet analysis for gigabit rate network traffic. Experimental test results indicate that our SB-NIDS prototype implementation on relatively low clock rate embedded processing platform performance is approximately 1.7 times better than Snort executing on a general purpose processor on PC when comparing processor cycles rather than wall clock time.

Contents

List of Figures

List of Tables

Abbreviations

AC	**A**ho-**C**orasick
ARP	**A**ddress **R**esolution **P**rotocol
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASIP	**A**pplication-**S**pecific **I**nstruction-set **P**rocessor
BGP	**B**order **G**ateway **P**rotocol
BM	**B**oyer-**M**oore
BRAM	**B**lock **R**andom **A**ccess **M**emory
CAM	**C**ontent **A**ddressable **M**emory
CERT	**C**omputer **E**mergency **R**esponse **T**eam
CIAC	**C**omputer **I**ncident **A**dvisory **C**apability
CPU	**C**entral **P**rocessing **U**nit
DEHA	**D**ecision **E**ngine **H**ardware **A**ccelerator
DoS	**D**enial **o**f **S**ervice
DDoS	**D**istributed **D**enial **o**f **S**ervice
DPI	**D**eep **P**acket **I**nspection
ESMTP	**E**nhanced **S**imple **M**ail **T**ransfer **P**rotocol
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSL	**F**ast **S**implex **L**ink
Gbps	**G**iga-**b**it **p**er **s**econd
GPU	**G**raphic **P**rocessing **U**nit
GUI	**G**raphical **U**ser **I**nterface
HDL	**H**ardware **D**escription **L**anguage
HIDS	**H**ost **I**ntrusion **D**etection **S**ystem
HTTP	**H**ypertext **T**ransfer **P**rotocol
IC	**I**ntegrated **C**ircuit
ICMP	**I**nernet **C**ontrol **M**essage **P**rotocol
IDS	**I**ntrusion **D**etection **S**ystem
IP	**I**nternet **P**rotocol
IPS	**I**ntrusion **P**revention **S**ystem
IPsec	**I**nternet **P**rotocol **S**ecurity
IPv4	**I**nternet **P**rotocol **v**ersion **4**
IPv6	**I**nternet **P**rotocol **v**ersion **6**
IRC	**I**nternet **R**elay **C**hat
ISP	**I**nternet **S**ervice **P**rovider
LMB	**L**ocal **M**emory **B**us
MAC	**M**edia **A**cess **C**ontrol
MIT	**M**assachusetts **I**nstitute **o**f **T**echnology
MTU	**M**aximum **T**ransmission **U**nit
NIDS	**N**etwork **I**ntrusion **D**etection **S**ystem

NIDS	**N**etwork **I**ntrusion **P**revention **S**ystem
NPU	**N**etwok **P**rocessing **U**nit
OPB	**O**n-**c**hip **P**eripheral **B**us
OS	**O**perating **S**ystem
OSPF	**O**pen **S**hortest **P**ath **F**irst
OTN	**O**ption **T**ree **N**ode
P2P	**P**eer-**t**o-**P**eer
PCHA	**P**acket **C**apture **H**ardware **A**ccelerator
PMHA	**P**attern **M**atching **H**ardware **A**ccelerator
POD	**P**ing **o**f **D**eath
PCRE	**P**erl **C**ompatible **R**egular **E**xpressions
QoS	**Q**uality **o**f **S**ervice
RAM	**R**andom **A**ccess **M**emory
RISC	**R**educed **I**nstruction **S**et **C**omputer
RPC	**R**emote **P**rocedure **C**all
RTN	**R**ule **T**ree **N**ode
SB-NIDS	**S**ignature **B**ased-**N**etwork **I**ntrusion **D**etection **S**ystem
SB-NIPS	**S**ignature **B**ased-**N**etwork **I**ntrusion **P**revention **S**ystem
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
SIP	**S**ession **I**nitiation **P**rotocol
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol
SoC	**S**ystem **o**n **C**hip
SoPC	**S**ystem **o**n **P**rogrammable **C**hip
SPI	**S**tateful **P**acket **I**nspection
SRAM	**S**tatic **R**andom **A**ccess **M**emory
SRT	**S**nort **R**ule **T**ree
SSL	**S**ecure **S**ocket **L**ayer
TCP	**T**ransmission **C**ontrol **P**rotocol
TFN	**T**ribe **F**lood **N**etwork
ToS	**T**ype **o**f **S**ervice
TTL	**T**ime **t**o live
UDP	**U**ser **D**atagram **P**rotocol
US-CERT	**U**nited **S**tates-**C**omputer **E**mergency **R**eadiness **T**eam
VoIP	**V**oice **o**ver **I**nternet **P**rotocol

Chapter 1

Introduction

This thesis is about optimisation of Signature-based Network Intrusion Detection System (SB-NIDS) packet analysis speed. A complete SB-NIDS prototype is presented, developed using hybrid hardware-software embedded processing platform.

1.1 Background and Problem Overview

One of the effective ways to secure computer networks from the attacks is the network defence software technology such as Network Intrusion Detection System (NIDS). NIDS has been widely adopted in the business and government sector to secure computer networks by detecting different kinds of network attack as well as detection of illegal access to confidential data and resources. Although NIDS is a good network defence software, on high data rate networks such as gigabit rate its performance is poor. It is unable to analyse all traffic as network packets arrive faster than NIDS packet analysis speed. Consequently, NIDS packet buffer fills very quickly and force it to drop some packets without analysis in order to make more space in buffer for new packets. For example, SB-NIDS software package Snort when executed on Intel Xeon Dual-Core 2.0 GHz general purpose processor and test with network test data which requires 175 concurrent Transmission Control Protocol (TCP) connections is only able to analyse network traffic up to 566 Mbps throughput [2, 3]. The main reasons for slow packet analysis rate are the complexity of network packet analysis operations and frequent memory accesses. These operations typically involve bit masking, bit comparison, bit shifting that general purpose processor instruction sets do not support efficiently. Also frequent memory accesses on these loosely coupled processor architectures consume a relatively high number of CPU clock cycles (100s) if a cache miss occurs (Section 6.3).

Various attempts have been made to optimise SB-NIDS packet analysis speed (Chapter 3). Most of the current state of the art solutions optimise only specific computationally intensive parts or components of SB-NIDS software (Section 3.5). Some solutions use clusters of processors and high performance embedded processing platforms in order to optimise a complete SB-NIDS application (Section 3.4.1). Commercial NIDS solutions based on embedded processing platforms use high specification embedded processing hardware. They claim to provide a complete SB-NIDS solutions with a support of up to 10 Gbps network throughput (Section 2.4.5). Still a great deal of work needed to be done to support NIDS packet analysis throughput preferably over 10 gigabit rate as network data rates continuously increasing such as recently approved IEEE 802.3ba standard which supports 40 Gbps and 100 Gbps transmission rate [4].

1.2 Solution Synopsis

A prototype SB-NIDS using hybrid hardware-software embedded processing platform was developed to enable high speed packet analysis. To support this effort, an open source and widely used SB-NIDS software package Snort was used. An execution analysis of Snort was carried out using software profiling tools in order to identify bottlenecks in packet analysis. Based on the profiling results, the most suitable embedded processing platform was selected. This embedded platform is a hybrid hardware-software processing solution having tightly coupled hardware architecture to enable low clock cycle accesses to hardware peripherals such as Network Interface Cards (NICs) and memory. It also allows hardware accelerator development in Field Programmable Gate Array (FPGA) and multi-processing using MicroBlaze soft processing cores. One of the main goals of hardware accelerator development is to offload any computationally significant operation of a software from a CPU to FPGA. FPGA provides parallelism, pipelining and bit-level processing facility, and has great potential to reduce and/or remove performance bottlenecks in SB-NIDS software. Multi-processing facility can be applied to SB-NIDS packet analysis process in order to improve the overall SB-NIDS processing efficiency although this is not investigated in this thesis.

Prototyping and optimisation was carried out in stages. Initially, Snort was ported to the embedded processing platform. This involved Snort software architecture restructuring in order to successfully map and execute to new processing platform which is based on Xilinx MicroBlaze soft-core processor and FPGA. This prototype is called as Manchester Metropolitan University-Snort I (*MMU-Snort I*). MMU-Snort I performance on new processing platform then evaluated to identify any improvement or bottlenecks in packet analysis processes. Test results showed that prototype SB-NIDS or MMU-Snort I

execution speed has improved when compared with the CPU clock cycles of the original Snort software package when executed on a PC with a general purpose processor.

In the next stage pattern matching algorithm performance was optimised. This involved development of Pattern Matching Hardware Accelerator (PMHA) and its integration into Snort prototype or MMU-Snort I. This resulted in the second prototype called *MMU-Snort II*. PMHA provides high throughput and low memory pattern matching solution using a Bloom filter data structure based approach which allows quick lookup of keywords or patterns [5]. The size of the attack signatures is significantly reduced to the extent that the whole attack pattern represented as Bloom filter is stored to FPGA on-chip memory/Block Random Access Memory (BRAM) for quick checking of signature presence in packet payload data. Aditionally, full database of patterns is stored in off-chip Synchronous Dynamic Random Access Memory (SDRAM) for further signature checking in case a signature stored in FPGA BRAM is found in a packet during first stage of pattern lookup. This is because Bloom filter search approach can provide false positive matches for pattern lookup. The integration of PMHA into MMU-Snort I required further design changes, resulting in an optimised and integrated PMHA for Snort on the same processing platform. In the last stage, PMHA was further optimised to search efficiently longer patterns ($>$ 64 Bytes) and for fast pruning of Bloom filter false positive matches which resulted in *MMU-Snort III* prototype.

In summary, investigation and development of complete SB-NIDS prototype using hybrid hardware-software processing platform is presented here, with a focus on the added benefits of throughput and reduced memory requirements. A full description of the architectures and algorithms is presented in this thesis. In addition, a detailed performance analysis and experimental results are used to demonstrate the suitability of hybrid hardware-software platform for SB-NIDS execution, higher throughput support and smaller memory requirements.

1.3 Aims and Objectives

The aim of this research project is to develop a series of algorithms and hardware architectures for SB-NIDS in order to optimise its packet analysis speed.

1.3.1 Objectives

In order to achieve this aim; following research objectives have been identified:

- Design and implement SB-NIDS prototype that perform network packet analysis at higher speed than Snort on general purpose processor.
- Evaluate SB-NIDS prototype using publicly available data for SB-NIDS testing and identify packet analysis speed improvement and/or any performance issues.
- Gain efficiency and/or improve SB-NIDS packet analysis speed by offloading pattern matching from CPU to FPGA that search one of the largest attack patterns preferably at gigabit data rate.
- Use Snort application specific knowledge (Section 5.2.2) to improve pattern matching (PMHA) performance further by reducing number of pattern search per packet.
- Evaluate PMHA implementation using publicly available test data for SB-NIDS in order to to verify its performance improvement.

1.4 Contributions and Claims

The primary objective of this effort is the development of SB-NIDS prototype solution using one of the viable embedded hybrid hardware-software processing platforms. The main challenge faced in the development on this platform is the design and implementation of a full SB-NIDS while improving the performance needed by the state of the art networks of today. This embedded processing platform is flexible as it not only allows the execution of complete software applications such as SB-NIDS on an embedded processor but also supports software optimisation, and enables the offloading of processing from a CPU to FPGA hardware and multi-processing with multiple processing cores. It also enables overall improvements in packet analysis speed of SB-NIDS due to tightly coupled hardware architecture in which hardware peripherals like Ethernet network interface have significantly lower latency access to network data and the CPU performs relatively fast off-chip memory accesses requiring tens of clock cycles in comparison to hundreds of clock cycles on general purpose processor architectures. The platform memory hierarchy arrangement in low-to-high clock cycles memory access sort order are, FPGA BRAM, off-chip Static Random Access Memory (SRAM) and off-chip Synchronous Dynamic Random Access Memory (SDRAM). This allows further improvements in the SB-NIDS performance by storing frequently access data such as network attack patterns or signatures to low latency access memory. Overall the processing platform features is found to be viable to implement the SB-NIDS. Also, flexible hybrid hardware-software processing and scalability in terms of multiple processing cores are ideal for further optimisation of other SB-NIDS components that are bottlenecks on high speed packet analysis. The idea presented in this thesis is a prototype SB-NIDS solution

that is intended to demonstrate the suitability of hybrid hardware-software processing for SB-NIDS execution, optimisation and further research and development.

Initially a study of Snort SB-NIDS software architecture was carried out in detail to understand the packet analysis mechanism and processing requirements, this study was then used to restructure Snort architecture for porting and optimisation on hybrid hardware-software processing platform, and in this way a novel SB-NIDS prototype architecture called MMU-Snort I on this platform was devised. In the next stage MMU-Snort II was developed which involved the development of special PMHA and algorithm suitable for low memory hardware that can also be easily integrated with MMU-Snort I. This effort tempts the development of a algorithm that reduces the amount of large memory required to store the attack signatures and reduces the pattern matching computational time to enable high speed pattern search in payload. The primary benefit attained through this which also advances the state of the art, is the compression of the whole attack signatures that fit on-chip FPGA memory for performing quick filtering or lookup, less computations required for pattern search than other closely related state of the art solutions that were implemented using bloom filter based pattern matching technique, and a complete SB-NIDS prototype based on one of the effective and widely used open source SB-NIDS, which when further optimise in future using multiple processing core and also migrate to a recent and more advanced hybrid hardware-software processing platform will enable the high data rate network protection.

The strengths and weaknesses of the architecture and algorithms were analysed, and experimental results were obtained to determine the effectiveness of the implementation. The resulting contributions of this work are:

- A novel SB-NIDS prototype based on one of the effective and widely used open source SB-NIDS software package *Snort* prototyped on a hybrid hardware-software processing platform for further research, development and optimisation.
- A PMHA that compactly store 7876 unique attack patterns in 8 KB of FPGA BRAM using Bloom filter search approach that supports quick lookup of packet data for checking pattern presence.
- An application specific PMHA design which also integrated into prototype SB-NIDS that supports parallel pattern search in a packet payload up to 1.85 Gbps throughput.
- An inclusion of Snort application specific knowledge or logic to PMHA that reduces the number of packet data lookup in Bloom filter stored in FPGA BRAM for 45 % of SB-NIDS attack signatures [1].

[1] A JAVA program is written to count the percentage of application specific information.

- An adoption of technique to Bloom filter search approach that reduces the number of hash value computation required by the Bloom filter based search algorithm [6]. The PMHA computes only two hash value in this implementation which produces the same Bloom filter false positive rate as with ten hash values.

1.5 Thesis Outline

This thesis describes a solution to the problem presented in section 1.1. Chapter 2 is the detail discussion on network security issues and technologies. Chapter 3 is the summary and discussion of the state of the art SB-NIDS and pattern matching solutions. Chapter 4 is the proposed system architectures detail. Chapter 5 is the design and implementation details of SB-NIDS (MMU-Snort I) and PMHA (MMU-Snort II and MMU-Snort III). Chapter 6 presents the analyses and results which include comparison of all three prototypes performance in order to estimate performance improvements. This also include comparison of the results with the closely related state of the art PMHA. Finally, Chapter 7 summarises the findings and provides some insight into future directions.

Chapter 2

Background

> "Whoever thinks his problem can be solved using cryptography,
> doesn't understand his problem and doesn't understand cryptography
> (Arritributed by Roger Needham and Butler Lampson to each other)"

The statement above is also relevant for current state of the art computer security defence mechanisms such as Firewalls and Intrusion Detection System (IDS). Currently, no effective solution exists for all network security problems and in some situations defence mechanisms are themselves vulnerable to network attacks and/or their failure may lead to complete failure of a network. To understand the scale of network security problems, issues concerning network security threats and defence mechanisms are discussed in detail in this chapter. The current state of the art in network security technologies are presented in (Chapter 3).

2.1 Chapter Roadmap

The rest of the chapter is outlined as follows:

- In section 2.2, range of network security issues are discussed concerning two core problems: flaws in network protocol design with reference to the Internet Protocol suite (also known as Transmission Control Protocol/Internet Protocol (TCP/IP)) and vulnerabilities in software applications and Operating Systems (OS). Other network security threats (Malwares such as Viruses, Worms and Spywares) are discussed briefly to clarify and quantify the scale of network security problems.
- In section 2.3, core network defence mechanisms are discussed to illustrate counter measures deployed against common network security attacks.

- In section 2.4, the main issues concerning Intrusion Detection System (IDS) technology, limitations and performance are explained in detail. The discussion main focus is Signature-Based Network Intrusion Detection System (SB-NIDS) technology.

2.2 Network Security Issues

The increasing frequency of cyber crimes and computer hacking attacks such as Denial of Services (DoS) and phishing attacks are indicated by [7–10]. Further, a recent malware (Section 2.2.3) attack on the University of Exeter (UK) computer network resulted in the shut down of their entire campus network service [11]. Such problems have occurred in domains including banking, financial services and retail. Network security problems are compounded by a heavy reliance on Web for accessing everyday services such as bill payments and online shopping transactions. Consequently, web based services are an attractive targets for hackers whose goal is to steal personal financial details. The hackers themselves are competent, but a big part of problem is actually the "Internet" itself as explained by Professor Ross Anderson,

> "The Internet protocol suite was designed for a world in which trusted hosts at universities and research labs co-operated to manage networking in a co-operative way [12]."

Consequently, the Internet was not designed to support secure online services. Flaws such as easy to manipulate key packet values (Internet Protocol address, Port number etc) in the "Internet protocol suite (TCP/IP)" hinder the provision of effective mechanisms to deal with network security issues. Further, software application vulnerabilities enables hackers to install malware on computers in order to create large distributed collections of hacked machines (botnets) that are deployed to launch Distributed Denial of Service (DDoS) attacks on computer systems or services (Figure 2.1).

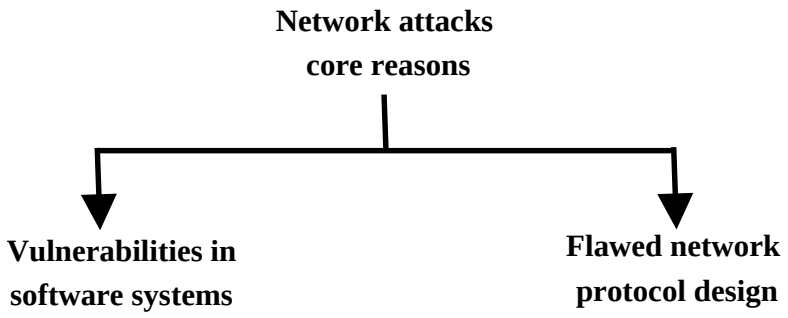


FIGURE 2.1: Two core reasons of network attacks

2.2.1 Flawed Internet Protocol Design

Network protocols such as IP, TCP, Simple Mail Transfer Protocol (SMTP), and Hypertext Transfer Protocol (HTTP) are the core protocols for packet based communication over the Internet. These protocols and others such as Peer-to-Peer (P2P) Protocol, Voice Over Internet Protocol (VoIP), Session Initiation Protocol (SIP) have lead to the development of online network services such as file sharing, electronic media streaming and financial payment systems. The secure provision of such services is challenging because they are built on top of IP protocol which is itself flawed. These protocols do not directly provide authenticity or confidentiality protection. Thus, knowledgeable users can exploit basic sniffer programs to inspect network packets and even to manipulate critical protocol features inside packets such as IP addresses, ports and payload data values. Protocol based network attacks exploit authenticity and confidentiality issues and may involve manipulation of packets in order to launch:–

- Denial of Services (DoS) and Distributed Denial of Service (DDoS) attacks
- Forgery attacks
- Session Hijacking attacks

In the following sections, protocol based attacks are discussed in order to demonstrate the exact nature of problems and flaws with Internet protocols.

Denial Of Service (DoS) Attacks

Definition: DoS is an attack on critical network resources such as routers, OS, application servers, network links aimed at disrupting their normal function.

DoS attacks can bring down critical network resources and normally results in communication disruption and service access denial to legitimate hosts.

Classification: Typically DoS attacks strive to deplete available processing and network bandwidth resources which Hussain et al suggested to classified into: *software exploits* and *flooding attacks* [13] (Figure 2.2).

Software exploits are a direct attack on critical network computing equipments (Routers, Switches, Server machines etc.) whereas *Resource depletion* attacks aim to deplete computational processing resources. A *TCP SYN DoS* attack sends a succession of SYN requests to a server (Section 2.2.1) [1]. The goal of this attack is to cause memory

[1]TCP SYN packet used in three way TCP handshake process between two machines for establishing TCP connection.

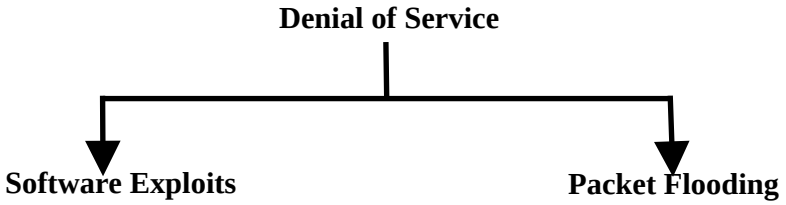


FIGURE 2.2: Denial of Service classification

and processing resource depletion in order to prevent new legitimate requests for TCP connections to open and/or for those currently open to fail to make progress on the machine under attack. A *flooding or bandwidth depletion attack* is an attempt to deplete network link bandwidth by sending a high volume of network packets. In this type of attack, an attacker floods the target network with network packets that completely occupies the network bandwidth and does not let the legitimate user to get connected to the network (Figure 2.3).

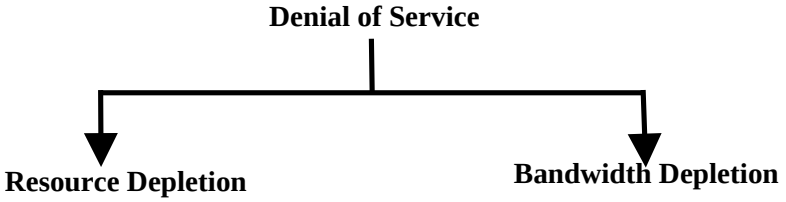


FIGURE 2.3: Denial of Service second classification

Network Protocols in DoS Attacks

DoS attacks are typically carried out using lower level network protocols: TCP, Internet Control Message Protocol (ICMP), User Datagram Protocol (UDP), and IP. Some of these protocols used for DoS attack identified by Computer Emergency Response Team (CERT) are [14]:

- TCP attack
- ICMP Ping of Death attack
- UDP flood attack

DoS attacks are not limited to this list of protocols but can also be carried out using IP packet by exploiting fragmentation and reassembly feature of IP packet.

DoS Attacks Using TCP

TCP is a connection oriented session level protocol for reliable duplex communication over the Internet. Many web services on the Internet utilise TCP. Such web services are hosted on server machines with potentially hundreds and thousands of client machines simultaneously connected via TCP protocol. Server machines must have high computational performance and large size main memory in order to serve each and every client request efficiently and to maintain TCP connection state. Servers are very vulnerable to attack as it is easy for hackers to launch DoS attacks exploiting TCP packet header flags bits (SYN, RST).

The two widely known DoS attacks carried out by the manipulation of TCP flags are: *TCP SYN attack* and *TCP Reset attack* [15, 16] (Figure 2.4).

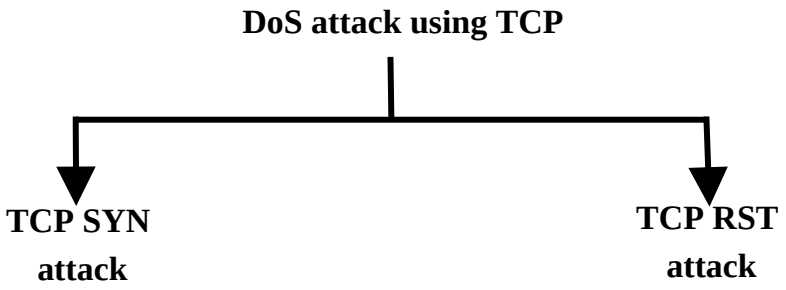


FIGURE 2.4: Denial of Service attacks using TCP

TABLE 2.1: Summary of DoS attacks using TCP

Attack	How	Effect	Remedy
TCP SYN	Attacker sends repeated TCP SYN packet with forged source IP address to target machine.	Server main memory becomes full due to pending connection request.	Software Patch
TCP RST	TCP RST packet with forged IP address is send by attacker to the target machine.	Drop connection that disrupts the communication.	No Solution

TCP SYN Attack: TCP protocol specification specifies one of the main purposes of TCP SYN flag is for establishing connection between two machines. This is carried out using a three way handshake process involving (SYN, SYN-ACK, ACK) packet exchanges. Hackers exploit this connection establishment process in order to carry out DoS attack. The attack involves sending a series of TCP packets to the target machine for connection request with a TCP SYN flag set and forged source IP address. The target machine on receipt of every TCP SYN packet must allocate space in the SYN queue in system memory and acknowledge the request by sending back a TCP packet with ACK flag set to the source machine IP address. The source machine will never acknowledge this message because source IP address is forged by the attacker. The result is that the

target system SYN queue can become full and force the target machine to reject further connection requests. Figure 2.5 shows a normal TCP connection establishment process and TCP SYN attack.

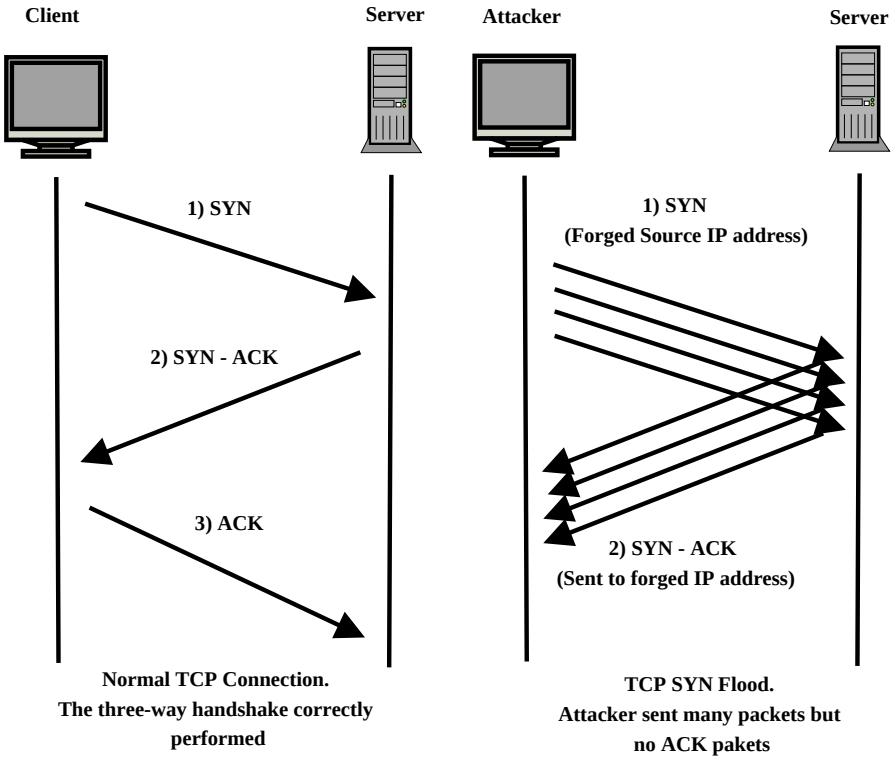


FIGURE 2.5: Normal TCP connection and TCP SYN flooding attack

TCP SYN is one of the classic examples of using the TCP protocol to attack network resources. However, this attack can be fixed by patching the OS (SYN cookie fix) or a defence mechanism can be used to monitor an unexpected number of flooded request (Section 2.3).

TCP RST Attack: The purpose of the TCP RST flag is to immediately terminate an established TCP connection between two machines. If a packet with TCP RST flag set to '1' received by the destination system then it immediately terminates TCP connection without any further exchange of TCP packets. The RST flag option is also exploited by attackers to launch DoS attack as examined for the first time in 2003 by Watson [16]. According to Watson's description, an attacker closely monitors TCP communications across networks and gathers vital information concerning host IP addresses and ports of current participants in TCP connections. On obtaining this information an attacker launches an attack by creating a TCP packet with a participants host IP address and ports along with TCP RST flag set and sends this packet to any of the two participants' hosts resulting in an instant connection termination and DoS effect.

DoS Attack Using ICMP

ICMP is a connectionless protocol and an integral part of IP suite. The ping command uses ICMP for troubleshooting network communication issues and gathering network details about hosts and open ports. The information gathered can be used to launch *ICMP DoS attack*. Two famous DoS attack carried out using ICMP protocols are the *Smurf attack* and *Ping of Death* attacks [17, 18] (Figure 2.6).

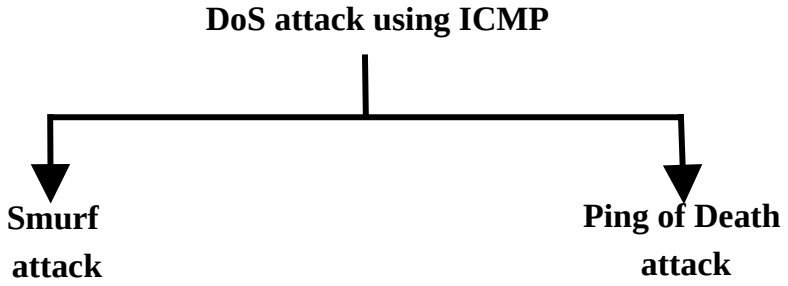


FIGURE 2.6: Denial of Service attacks using ICMP

TABLE 2.2: Summary of DoS attacks using ICMP

Attack	How	Effect	Remedy
Smurt attack	Attacker sends ICMP ping packet with target machine IP address set as source IP to the broadcast address.	Network bandwidth depletes due to packet flooding as well as processor resources exhausts.	Protocol fix proposed in August 1999
Ping of Death	Attacker sends oversize ICMP ping packets to the target machine.	Processors resource depletes due to operating system problem dealing with oversize packets.	Operating System patch

Smurf Attack: uses an ICMP echo request/reply (ping) packet. The attacker must acquire network information concerning particular IP addresses of hosts. The attacker then spoofs[2] ICMP echo request packets to be from one or more valid hosts (victims) and sends a high volume of packets to the network broadcast address. Now every host on a network receives an ICMP packet and they reply back to the spoofed IP address of the victims (valid hosts). Now each victim who originally did not send any ICMP packet will receive ICMP echo reply from all hosts on a network and this depletes network bandwidth and may overwhelm the victim machine causing DoS effect. Smurf attack is shown in figure 2.7.

Figure 2.7 shows only one ping request in smurf attack. A technical countermeasure was proposed in August 1999 to the protocol standard that ICMP echo requests sent to the broadcast address are no longer replied to by default [19]. Router configuration can also be used to prevent the propagation of broadcast packets to other subnetworks.

[2]Spoofing is the creation of packets with forged or incorrect source address.

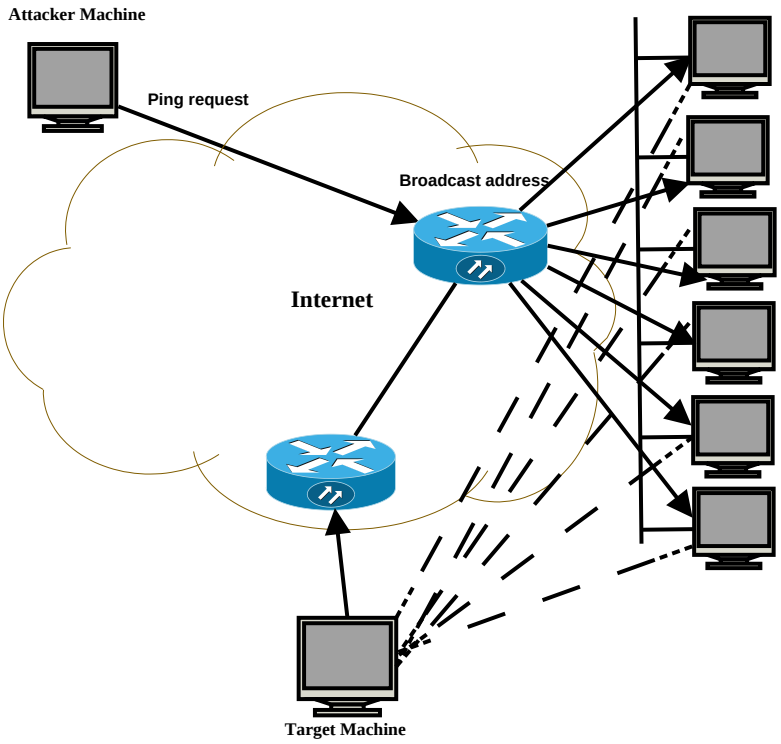


FIGURE 2.7: Smurf attack

Ping Of Death Attack: Another kind of DoS attack using ICMP is also carried out using ICMP echo request/reply packet or ping packet called *Ping of Death* (POD) DoS attack. An attacker sends a high volume of malformed or oversize ICMP ping packets[3] up to the allowed maximum IP size of 65535 bytes. These packets are then fragmented by the transmission network. The victim machine receiving such fragmented packets tries to reassemble them and a buffer overflow can occur causing a system crash or reboot. This attack is no longer effective as most operating systems are patched to deal with malformed ping packets by allocating a significant amount of memory and via the incorporation of buffer overflow checking code.

DoS Attack Using UDP

UDP is a connectionless transport layer protocol. Thus, it is possible to send any type of UDP packet to any machine on any ports without informing the recipient or handshaking. *UDP port DoS attack* is a DoS attack based on UDP [20].

UDP Port DoS Attack: here, an attacker sends a series of UDP packets to an IP address of a victim machine on a specific port or a set of random ports. For every UDP packet received by a victim, its OS tries to determine which application provides services on this port. If the victim machine is not running any application for the

[3]Ping is 56 bytes in size and 84 bytes when IP header is considered.

requested service on a port then it replies with an ICMP destination port unreachable packet indicating there is no service available. If the attackers send UDP packets in large volume then the overhead of triggering repeated ICMP destination port unreachable reply packets can overwhelm the victim machine processing and it may even deplete network bandwidth.

This attack can easily be stopped by analysing the network traffic using network defence technology such as NIDS with correct packet filtering policies (Section 2.3).

DoS Attack Using IP

IP is widely used protocol and is compatible to work with different types of communication standards such as Ethernet, Fiber Distributed Data Interface (FDDI) etc. IP allows applications to send up to 65535 bytes in a single packet (including 20 bytes packet header). When a packet of such size is sent over a network it is often fragmented into multiple packets because of Maximum Transmission Unit (MTU)[4] packet/frame transmission limits. All fragmented packets must be reassembled at the destination system before passing upward to the application layer. The memory and computational requirements of reassembly can be used to cause DoS effect such as *Ping of Death* DoS attack. Others DoS attacks caused by manipulation of IP protocol are *Land attack* and *Teardrop attack* [21] (Figure 2.8).

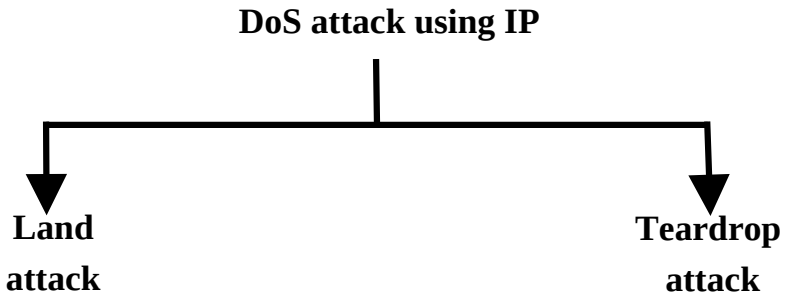


FIGURE 2.8: Denial of Service attacks using IP

Land Attack: here, attackers manipulate the IP packet source field by copying the destination IP address into the source IP address field in order to trick the destination machine into sending packets to itself to crash the machine and cause DoS effect.

Teardrop Attack: exploits a bug found in OS fragmentation and re-assembly code that improperly handles overlapping (and hence malformed) IP fragments. An overlapping IP fragment occurs if two or more IP fragments have offsets indicating that they overlap each other in position within the unfragmented IP datagram. An attacker sends a series

[4]Typical MTU for Ethernet is 1500 bytes.

Table 2.3: Summary of DoS attacks using IP

Attack	How	Effect	Remedy
Land attack	Attacker flood the target machine with IP packets with destination IP address of machine copied into source IP address field.	Target machine sends the packet to itself that exhausts the processor resources.	Operating System patch
Teardrop attack	Attacker sends series of overlapping IP fragments to victim machine that exploits bugs in OS fragmentation and reassembly code.	Processors resource depletes due to operating system problem with overlapping IP fragments.	Operating System patch

of overlapping IP fragments to a target machine and this cause system crash and DoS effect.

These two attacks can easily be stopped with proper configuration (OS patch) and/or by the deployment of defence mechanism such as firewalls or NIDS (Section 2.3).

Distributed Denial Of Service (DDoS) Attack

Definition: DDoS uses multiple compromised host systems to mount a coordinated attack on critical network resources in order to cause DoS effects.

Explanation: In DDoS attack there is typically a master host or node controlling a number of slave machines to initiate or stop an attack. The master node is in direct control of an attacker that has taken control of slave machines using *DDoS agent software*. DDoS agent software is installed into victim machines either by exploiting a vulnerability (Section 2.2.2) in software systems or by tricking the user into installing the agent software. There may be more than one master node controlled by the attackers that are used to manage the distributed network of slaves. An attacker usually instructs master nodes to launch an attack using active slaves and direct them to simultaneously launch an attack. This DoS attack when launched with a number of nodes is called *DDoS attack*. Network topology of DDoS is shown in figure 2.9.

According to Computer Incident Advisory Capability (CIAC) (Renamed to Department of Energy-Cyber Incident Response Capability (DOE-CIRC) [22]), the first such kind of DDoS was seen in the summer of 1999 with the introduction of DDoS attack tools [23]. DDoS exploits the weaknesses of common Internet protocol and launches DoS attacks (TCP SYN flood attack, UDP flood attack, ICMP flood attack etc.) in a distributed manner in order to quickly deplete processing and bandwidth resources of target machines and network. First successful DDoS attacks noted in the year of 2000 when Yahoo

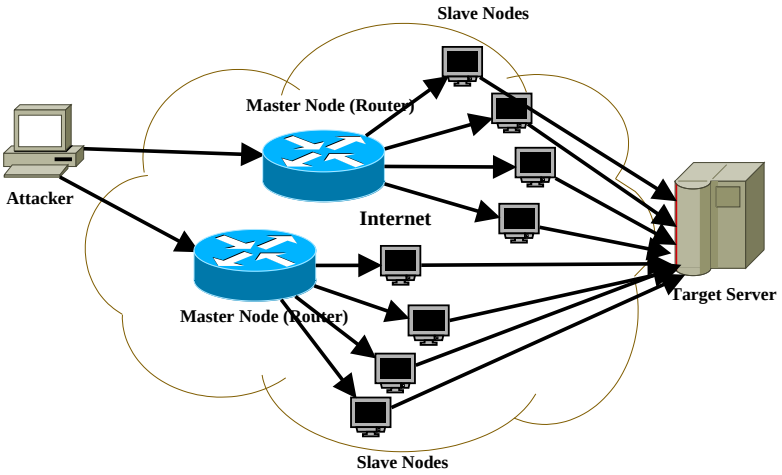


FIGURE 2.9: Network topology of DDoS

website was attacked which resulted in DoS [24]. Table 2.4 has the summary of some common DDoS attack tools.

TABLE 2.4: Common DDoS attack tools

Tools	Protocols	Description
Trinoo (aka Trin00)	UDP	This tool sends out a large number of UDP packets to the victim.
The Tribe Flood Network (TFN)	ICMP, TCP and UDP	This tool is able to attack victims with ICMP flood, SYN flood, UDP flood and Smurf attacks.
Stacheldraht	ICMP, TCP and UDP	This tool combines the features of Trinoo and TFN with encryption support.
Trinity	TCP and UDP	This tool uses Internet Relay Chat (IRC) for launching UDP or TCP flood attack.
Tribe Flood Network 2K (TFN2K)	ICMP, TCP and UDP	Successor to TFN. This tool uses TCP, UDP, ICMP or a Smurf packet flood to target the victim.
Shaft	ICMP, TCP and UDP	This tool uses TCP, UDP and ICMP packets or all three at the same time for flooding victim.
Omega	ICMP, TCP, UDP and IGMP	Similar to Shaft. This tool use TCP, UDP, ICMP, IGMP or mixture of protocols to flood the victim.

Forgery Attacks Using Protocols

The basic problem is that the core Internet network protocols provide no significant authenticity or confidentiality protection. Network packets can easily be intercepted and manipulated to forge or alter IP addresses in order to create spoofed packet containing false values. Forgery is usually carried out to misrepresent another computer system, or for attackers/hackers to conceal their identity by forgery which is clearly illegal.

However, packet manipulation and forgery are correct or, atleast are accepted according to IP protocol specifications in order to support functionality such as legitimate remote access to electronic services whose access is restricted via source IP address. Most DoS attacks discussed in previous section and all in this section are carried out with spoofed addresses.

Table 2.5: Forgery using Network protocols

Protocol	Description
SMTP (Email address)	An email address to represent email sender/receiver identify can be forged to carry out spamming or phishing attack.
IP (IP address)	Most of the attacks on networks and IT infrastructure use IP address forgery to misrepresent or hide real identity of attackers.
MAC (MAC address)	MAC address manipulation on LAN used to hide or misrepresent attacker identity to carry out attack such as session hijacking.

Forgery Attack Using SMTP

Simple Mail Transport Protocol (SMTP) is the standard protocol for sending electronic mail (E-mail) across IP networks. SMTP is a simple text based protocol where client and server exchange string commands for connection establishment, authentication, and sending of mail data over reliable transport layer (TCP) to recipient mail servers.

SMTP has no real security mechanism as it is created on the idea of co-operation and trust like other lower level protocols. SMTP allows open mail relay. This means an SMTP server can be configured to allow anyone on the Internet to send e-mail through it, not just mail destined for, or originating from, known users. This significantly contributes to the large security problem of current time known as *E-mail spamming*.

E-mail Spamming: Spammers usually use forged email addresses in order to pretend their E-mail comes from a real company (in some cases invalid email addresses are deployed to conceal identity). The goal of Spam is to trick a user into releasing sensitive information such as Bank details or to lure them into the inadvertent installation of a trojan horse or virus (Section 2.2.3) when downloading an apparently useful program. Basic SMTP has no mechanism to authenticate users and/or to verify forged sender email address in packet headers and it will assume that any correctly formatted email address is valid. Email phishing attacks are carried out using forged email addresses.

Solution: There is no effective way to stop spam emails. Although proper configuration of SMTP server is recommended and there should be a single point of entry for connecting to an SMTP server usually through some kind of defence technology so every connection can be monitored (Section 2.3). The SMTP extension protocol called

Enhanced SMTP (ESMTP) provides user authentication based on usernames and passwords.

Forgery Attack Using MAC

Medium Access Control (MAC) is a link layer level protocol of TCP/IP that allows multiple devices to be connected to a shared physical communication medium. The data communication between machines in this shared medium is supported mainly with the help of a 48 bit unique MAC address (also known as a hardware address). This address is used in a shared communication medium to identify the individual machines.

MAC Attack: Attacker and victim machines should be on the same subnet for MAC based forgery attack. An attacker creates a spoofed packet with a forged MAC address to misrepresent another computer system. The main aim of the attack is to take over a victim's communication with another computer on a subnet exchanging sensitive data and information. There is no defined mechanism to detect MAC address forgery due to the lack of confidentiality mechanisms in network protocol. Figure 2.10 shows the forgery attack carry out using Address Resolution Protocol (ARP).

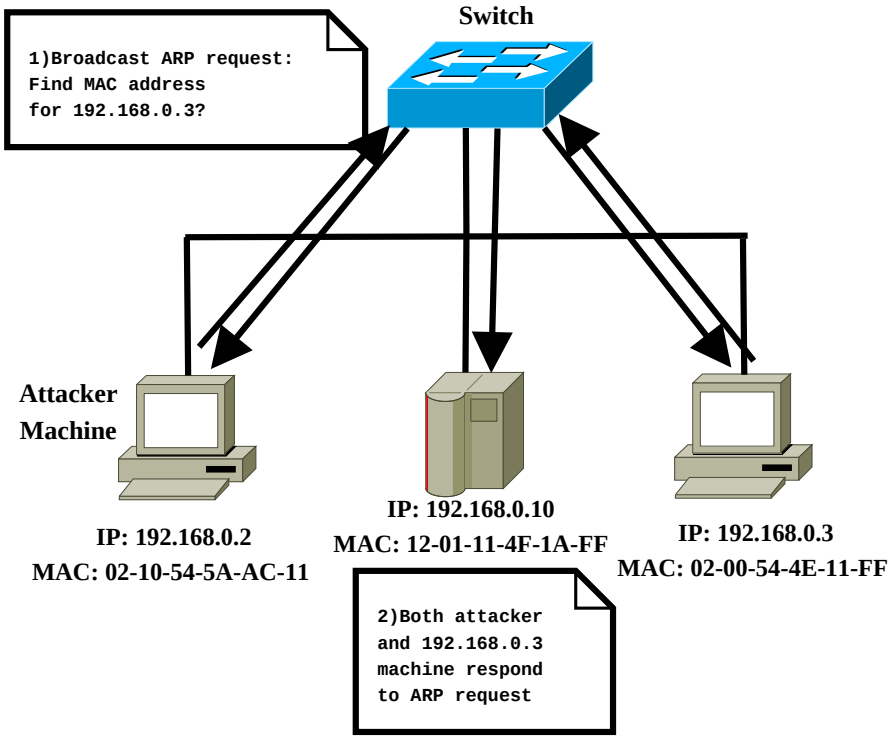


FIGURE 2.10: MAC address forgery attack also known as ARP spoofing

Forgery Attack Using IP

Most of the attacks discussed until now forged addresses such as MAC address, IP addresses or email address in order to create spoofed network packet for launching different types of complex network attacks such as DoS or DDoS. Some network configurations also provide access to network services on resources based on IP or MAC address authentication. If an attacker forges these addresses then it can gain access to services and information. There are other kinds of attack that carried out with the help of forged IP address such as *session hijacking and routing attacks.*

Session Hijacking Using Protocols

Definition: An unauthorised access to information or services in a computer system by infiltration through an already established connection between hosts is called *Session Hijacking.*

Explanation: Like most of the attacks discussed previously, spoofed packets with forged IP address are commonly used for session hijacking as well as for routing attacks. Attackers exploit flaws or weaknesses of protocols such as TCP, ICMP and Border Gateway protocol (BGP) in order to carry out session hijacking. Session hijacking and routing attacks involve direct intrusion into established network connections. The objective of these attacks is not just the disruption of communication between hosts but to actually change the routes used by a host or a router in order to eavesdrop on communication and steal vital sensitive information directly from network packets.

Session Hijacking Using TCP

TCP supports reliable duplex stream based communication. Packets are delivered in order using 32 bit sequence numbers in each and every TCP datagram.

Attack Description: TCP session hijacking attack alters TCP sequence numbers. An attacker must guess or intercept packets to get the correct TCP sequence number of a TCP session between two hosts. Then, with a TCP sequence number and a forged IP address of any other trusted host on the network, an attacker creates a spoofed TCP packet and sends it to the target/victim machine. Because of the correct sequence number the other host treats the packet as an established session packet and starts exchanging packets with an attacker machine and discloses sensitive information. The other machine that originally had an established session with the victim machine now has an invalid TCP sequence number and its TCP session with the host is invalid.

Session Hijacking Using Routing

Session hijacking can also be carried out by exploiting weaknesses in network routing table update procedures. The two protocols used for hijacking sessions are ICMP (Route Redirect message) and Border Gateway Protocol (BGP) [25, 26]. An attacker creates a spoofed packet with the forged IP address of a victim to misrepresent and inform the gateway or routers to update their route table with false routing information in order to redirect communication through a rogue or compromised gateway/router. In this way an attacker is able to successfully see all communication of a victim machine.

In this section a small number of protocol weaknesses and flaws were discussed along with variety of network attacks. A more comprehensive overview of network attacks and flaws in protocols is presented in the survey presented by Simon Hansman and Ray Hunt [27].

2.2.2 Vulnerabilities in Software

Vulnerabilities in softwares and bad host configurations provide additional opportunities for hackers to launch different types of network attacks. These vulnerabilities arise due to flawed programming or badly tested software releases. For example, it is relatively common for new or updated versions of libraries and applications to be released that are related to networking and operating system services. SysAdmin, Audit, Network, Security (SANS/MITRE) reported in 2009 about the top 25 dangerous coding errors where two of the errors led to more than 1.5 million US dollars of website security breaches during 2008 [28]. The common software flaws are unchecked user inputs potentially leading to buffer overflows and SQL injection[5], fundamental issues with OS user policies where possible unauthorised privilege escalation is possible with the potential for malware to execute commands on behalf of a hacker. Further, password management flaws can allow the user or root/Administrator passwords to be cryptographically weak and therefore easily compromised by hackers in order to gain unauthorised access. A key task for a network administrator is to be aware of current vulnerabilities in operating system and application software and to deploy updates or fixes as and when they become available. The next section discusses network attacks exploiting common vulnerabilities in software and is further evidence for the need for proper network configurations and defence mechanism for networks (Section 2.3).

[5]SQL injection refers to the unintentional direct execution of SQL statements by a program.

DoS Attacks Using Software Vulnerabilities

Some common DoS attacks carried out by hackers by exploiting software vulnerabilities are now discussed:

Buffer Overflow Attack: exploits flawed programming where a buffer can be overfilled and values are written into adjacent memory to the end of the buffer storage. This causes corruption of data which either crashes the application or it can be used to cause the execution of injected code in order to perform a malicious operation.

Crasher Attacks: causes the host systems to crash, leading to DoS due to a reboot. A popular example of crasher attacks exploiting OS vulnerabilities are *land attack* and *Ping of death* (Section 2.2.1).

2.2.3 Malicious Code

Malicious code is commonly referred as *Malware*.

Definition: Malware is a software program designed to perform malicious activities on a computer system without owner consent.

Explanation: Malware is deployed to cause damage to a computer system by an attacker. It can be used to take control of a computer system, to access sensitive information and to launch attacks on other computer system. Malware installation on any computer is difficult to detect and prevent. Advanced defence mechanisms sometimes even find it difficult to counter those new variants of malwares commonly known as zero day exploits (Section 2.4.6). However, certain combination of defence mechanism are effective but far from perfect in completely countering any new or zero-day malware attacks (Section 2.3).

Types Of Malware: Malwares can be self-replicating in order to propagate themselves either by attaching themselves from one computer file to another, or by emailing themselves to other machines using a victim's email client software address book. Examples of self-replicating malwares are computer *Viruses* and *Worms* (Figure 2.11).

Like self-replicating malwares, non-replicating malwares also performs malicious activities usually by fooling them installing on victims machine. Commonly known non-replicating malwares are *Trojan Horse*, *Spyware* and *Adware* (Bad Adware) (Figure 2.11). Other malwares include *backdoors*, *trojan downloaders*, *password stealers*, *Crimeware* and *Mobile malware*.

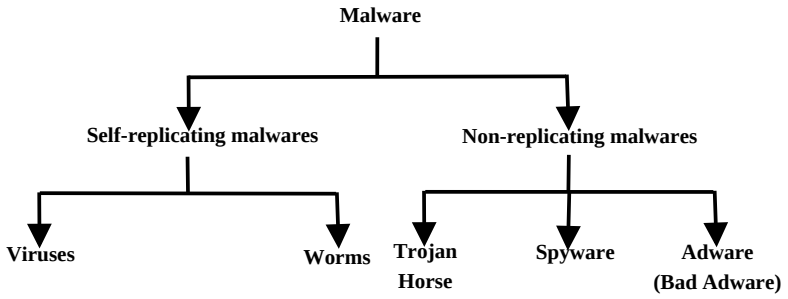


FIGURE 2.11: Types of malware

Crimeware: is a malicious software intended to yield financial benefits for an attacker using theft of personal information for fraudulent use, theft of trade secrets or intellectual property and spam distribution [29].

Mobile Malware: is a malicious software designed to infiltrate mobile devices without user consent. Security analysts believe mobile malware is a significant and large threat due to the exponential increase in number of mobile devices such as smart phones and netbook mobile devices with mobile internet connectivity [30].

Self-replicating Malware

Computer viruses and worms are self-replicating malwares that propagate and infect systems to perform malicious activity. A virus propagates by attaching itself to computer files. A worm propagates through a computer network without attaching itself to computer file.

Virus: The term computer virus first came to known in 1986. Fred Cohen's PhD thesis demonstrated how program code could propagate itself from one machine to another [31]. Table 2.6 summarises some common virus types [32].

The top four virus types in table 2.6 are easy to detect using virus scanner technology. However, the last three types of viruses use tactics that enable them to successfully avoid detection by virus scanners or other network defence mechanism such as SB-NIDS.

Worm: Computer worms typically cause more destruction to computer systems and networks than computer viruses. Computer worms are of two types: *Mass-mailing* worms and *Network-aware* worms [27]. Mass-mailing worms spread through emails. Example of mass-mailing worm is *Mellisa* that attaches itself to email for propagation. Network-aware worms are more sophisticated and destructive than mass-mailing worms. Network-aware worms look for known vulnerabilities in Internet hosts and try to gain access in order to compromise machines. Once a worm reaches a target machine it modifies critical operating system files to hide its identity and then attempts to propagate

Table 2.6: Viruses types and behaviour

Protocol	Description	Detection
File Infector	Mainly infects the program files (.EXE, .COM, .BIN etc). It may also infect script or configuration file.	Easy
Boot record infector	Infects system boot sector.	Easy
Multi-partite virus	Hybrid nature. Infects boot record as well as file.	Easy
Macro virus	Infects macro-enabled Microsoft office document.	Easy
Stealth virus	Stealth virus disguises itself to thwart detection by altering its file size, or concealing itself in memory.	Hard
Encrypted virus	It uses encryption to hide itself from virus scanners. Each time it infects it automatically encodes itself differently.	Hard
Polymorphic virus	Everytime this virus infects file it changes its signature.	Hard

to further hosts. Examples of this kind of worm is *SQL slammer* worm that exploits the known vulnerability in Microsoft SQL Server 2000 and Microsoft Desktop Engine.

Non-replicating Malware

There are other malicious codes or malwares which are intrusive, hostile and annoying. Unlike viruses and worms they are non-replicating and do not propagate themselves. Such malware often installs itself in a victim machine by tricking the user into believing them to be benign programs when actually they have a malicious purpose. Often such programs interfere with system settings and open backdoors for remote access to system resources. The two widely known non-replicating malwares are: *Trojan Horse* and *Spyware*.

Trojan Horse: Trojan Horse or a *Trojan* is a non-replicating malware that spreads by opening an email attachment or downloading and running a file from the Internet containing trojan code. Trojans appear to perform desirable functions but are used to compromise victim system security. Usually it changes system settings and allows hackers to remotely connect to a machine for malicious purpose. Desirable operations performed by hackers using trojans are file uploading and downloading, launching attacks such as DDoS and email spamming, other malware installations, data theft, remote screen viewing, rebooting the machine and resource hogging of hardisk space and processor computing power.

Spyware: A non-replicating malware that secretly installs itself in a victim's machine for the purpose of collecting data. Sometimes spyware come as part of regular software packages and are installed in a hidden directory. Spyware monitors victim's system usage and Internet surfing habits and sends the information over the internet to spyware owner

or hacker. Information collected using spyware is then used to target advertisements and to further tempt the user into installing other malicious malwares.

2.3 Network Defence Mechanism

This section discusses techniques and some state of the art network security technologies used to minimise and counter network threats.

2.3.1 Configuration Management

Configuration management aims to minimise network security issues through technically sound network organisation and by frequently applying application software updates and OS patches. Vulnerability scanner and configuration management software tools are available to automatically apply such updates and patches. Encryption is typically deployed to protect the transmission of sensitive information. Typical deployments include the IP layer using a framework such as IPsec or at the application layer using the Secure Socket Layer (SSL) protocol. Firewalls are typically used to secure a network from outside attack and to prevent subversive users from advertising unauthorised network services to the external internet beyond the firewall. Another strategy is Virtual Private Network (VPN) which securely sends private data between two sites or locations.

2.3.2 Firewall

A Firewall is a combination of software and hardware coupled with restrictions on network topology that is used to secure and limit network access. A typical firewall deployment is shown in figure 2.12. Firewalls are used to limit outgoing and incoming network traffic to ensure that it is well-formed (thus preventing spoofed packets entering or leaving the network) and legitimate. Standard networking equipment such as routers offer functionality such as packet filtering using rules to determine which incoming/outgoing network packets are allowed to pass or be dropped.

Types Of Firewall: Firewall comes in three main flavours as shown in figure 2.13, although variations of these basic organisations are also possible.

Packet Filter: A packet filter applies selective passing or blocking of network packets based on Open System Interconnection (OSI) layer 3 (IPv4/IPv6) and OSI-layer 4 (TCP, UDP, ICMP, and ICMPv6) headers. The most often used criteria of passing or blocking the packets are source and destination IP addresses (IPv4 and IPv6), source

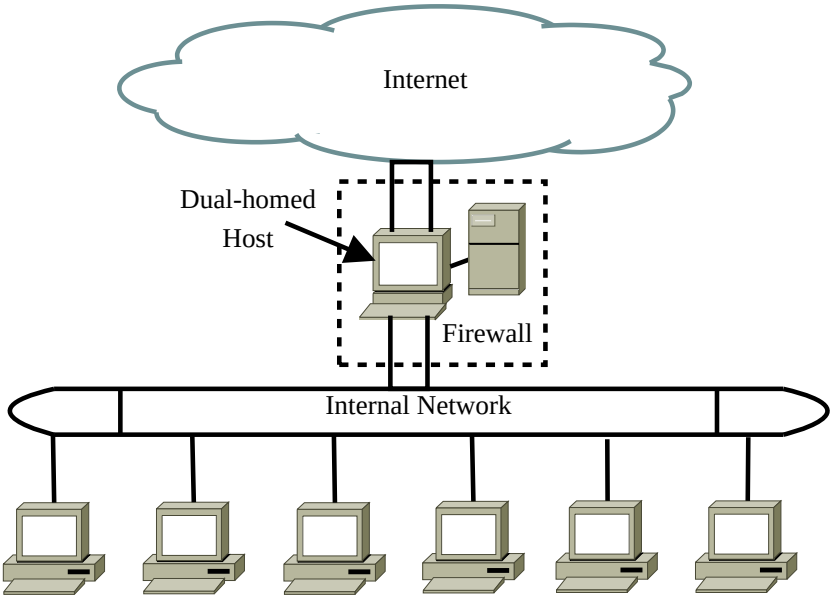


FIGURE 2.12: Firewall sitting between LAN and the Internet

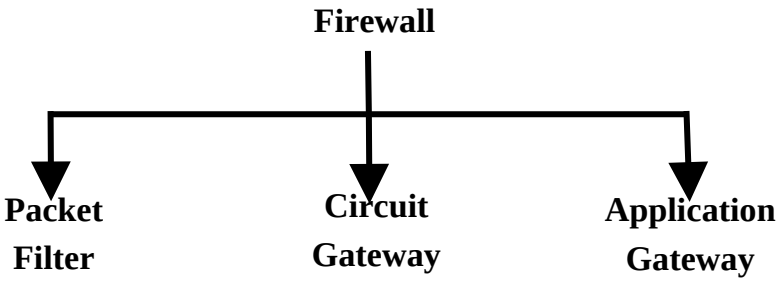


FIGURE 2.13: Types of Firewall

and destination ports (TCP, UDP, ICMP, and ICMPv6). Some packet filter also checks the ICMP codes along with other criteria to pass or block network packets.

Packet filters can detect packets that has spoofed or malformed IP addresses for launching different network attacks (Section 2.2.1). For example, packet filters can stop all the packets coming from outside the network with forged local network IP addresses in sender source packet header field to foil attacks. Packet filters cannot detect any protocol violation that is often used to disguise certain types of attack traffic because packet filters make no attempt to understand the packet payload data.

Circuit Gateways: It is a more complex firewall that can detect specific protocol violations. A circuit gateway monitors the transport level of OSI model (OSI layer 4 or TCP sessions). For example, TCP handshaking between host is checked to determine whether a requested session is legitimate. Also circuit gateway monitors the request/reply packets of established session for detecting protocol violation. For example, TCP SYN DoS attack can easily be detected by circuit gateways.

Circuit gateway is effective for session monitoring. However, it cannot prevent attacks carried out using malicious code encapsulated in packet payload data.

Application Gateways: operate at the application layer of the OSI model (OSI Layer 7), consuequently, they examine packet payload data to determine if the contents are well-formed and legitimate. Unfortunately, this means that specific code, or programs must be available to analyse the contents of multipacket messages in order to determine if the messages conform to the protocol specification and that they are well-formed. The computational and memory requirements of such gateways are significantly increased over other firewalls and they can become network bottlenecks that limit data-transfer rates. Application gateways can detect the spread of worm and computer viruses by comparing packet payload data with the signature of known malwares.

Currently firewall is virtually non-existent and is replaced by Internet Security systems. These systems are actually IDS or Intrusion Prevention Systems (IPS) which offers an effective means to detect and prevent the networks against complex kind of network attacks and malware outbreak by combining the power of all types of firewalls and malware scanners. The IDS technology is now discussed in more detail.

2.3.3 Intrusion Detection System

IDS is a security system for detecting attacks on a computer and network. IDS actively monitors events occuring on computer systems and networks and analyses them for the signs of suspicious activities. If IDS finds any suspicious activities then it logs event information and raises an alarm for the attention of Network Administrators as shown in figure 2.14.

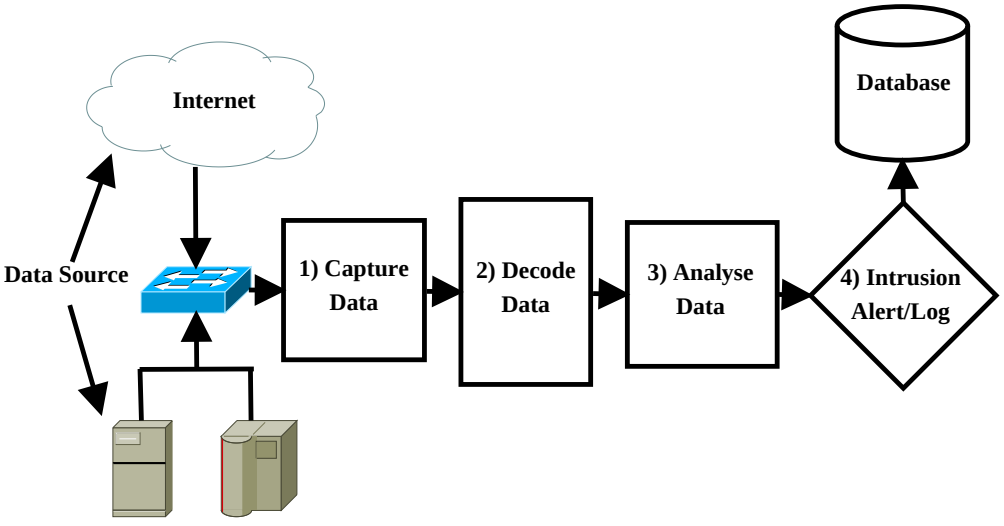


FIGURE 2.14: Typical IDS data analyses flow

2.4 Intrusion Detection System: An Indepth Analysis

IDS provides more effective security solution than firewall because they go beyond monitoring of network packets and even analyse OS events and log possible signs of suspicious activities. In this way an IDS detects those activities or attacks that cannot be detected with a firewall, such as trojans, spywares and other bad adwares.

2.4.1 Host Monitoring

Host monitoring IDS must carefully analyse every event occurring on a computer for the possible signs of suspicious activities. IDS must be trained to recognise suspicious event patterns, and or have knowledge of legitimate patterns of events. For example, typical computer authentication systems might allow a maximum of 5 consecutive failed login attempts for user access prior to the IDS on a host raising an alarm. Further, if any user performs suspicious operation on a computer then the host IDS should raise an alert for the attention of Network Administrators.

2.4.2 Network Monitoring

IDS when deployed for network traffic analysis, analyses all inbound and outbound network traffic and alerts network administrators on the discovery of potential network threats. Such IDS can identify network probes/scanning and attacks such as TCP SYN attack, viruses/worm outbreaks and attacks on software vulnerabilities.

An IDS is a passive monitoring system, since its primary purpose is to only alert the network administrators whereas an IPS, forcibly halts suspicious activity by for example blocking suspicious network packets, or halting activity due to suspicious events on a specific host in order to prevent an attack with the risk of potentially halting legitimate activity. Usually every IDS has an option to be configured as an IPS. For example, the IPS associated with Automated Teller Machine (ATM) may block or retain a card if incorrect pin is entered repeatedly.

2.4.3 Types of Intrusion Detection System

There are two types of IDS based on the type of events they monitor. The two types are called *Host IDS (HIDS)* and *Network IDS (NIDS)* (Figure 2.15). There is another kind of IDS that have combine capability of both IDS known as *Hybrid IDS*.

These IDS types may use two different types of intrusion detection techniques: *Signature detection* and *Anomaly detection*. IDS types and their detection techniques are now discussed.

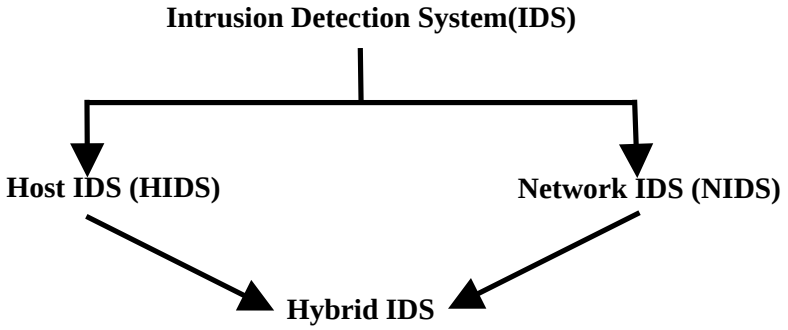


FIGURE 2.15: Types of IDS

TABLE 2.7: Comparison of IDS types

IDS Types	Analyse	Deployment
NIDS	Network traffic/data	Network points/backbone
HIDS	Host events/data	Individual machine

Network Intrusion Detection System

NIDS is a type of IDS that analyses network traffic for the detecting signs of malicious activities. NIDS is a complementary technology to firewall and detect attacks often missed by firewall. NIDS analyses incoming/outgoing network traffic, and can detect suspicious activities of spyware, adware, trojans and other malwares that are not detected by Firewalls. NIDS can inspect a range of protocol vulnerabilities from data link layer to application layer of TCP/IP protocol stack. It can also detect malicious code or malware in a network packet payload data using pattern matching commonly known as *Deep Packet Inspection*. Figure 2.16 shows the packet processing flow in NIDS.

Packets are capture from the network interface by NIDS's packet capture component. This packet is then passes on to the Packet Decoder component that stores data to memory for later analysis concerning the decoded packet protocol/payload information. The data analysis component analyses this decoded packet protocol data for any protocol anomalies and malware patterns. Finally, the intrusion alert component raises the alert and/or logs the events.

NIDS Deployment: A NIDS can be deployed on a point in a network where network packet inspection is essentially required. A large number of NIDS can be deployed either on a major network backbone to analyse every packet crossing through networks or it can also be setup at only specific network points analysing traffic directed to specific

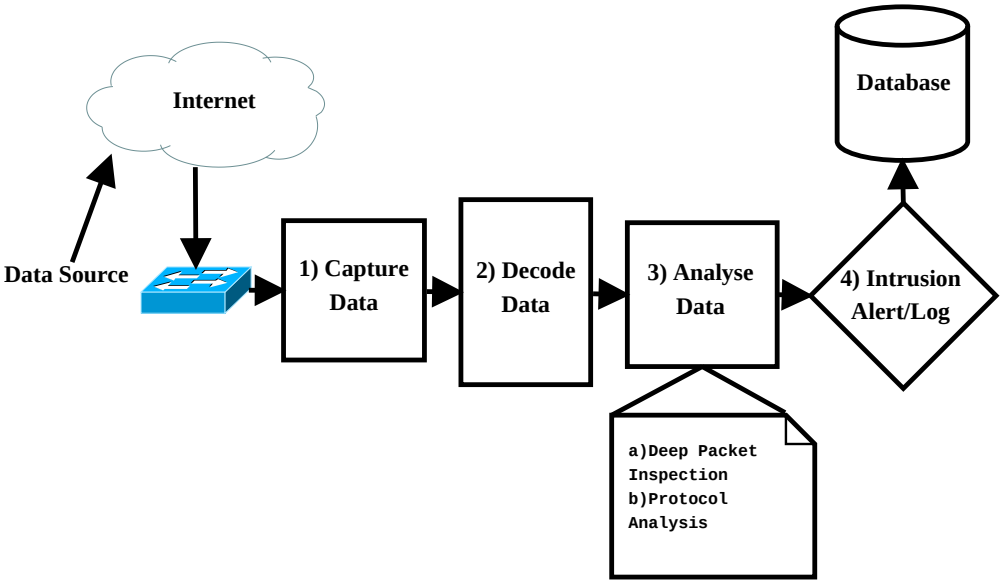


FIGURE 2.16: Typical NIDS data analyses flow

server or machines. A common strategy of deployment is a combination of firewall and NIDS. Whatever deployment position is decided, NIDS should not be accessible from anywhere on a network, so it is not assigned an IP address. However, it is normally accessible locally from inside the network for configuration and maintenance.

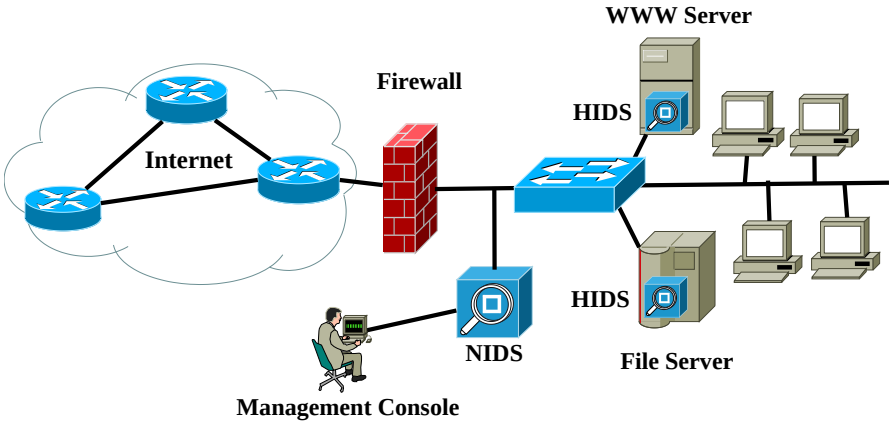


FIGURE 2.17: NIDS sitting between LAN and the Internet

Figure 2.17 shows NIDS deployment in which a firewall first filters network packets and remaining packets are analyse by NIDS. NIDS network interfaces are connected to the main network link via port mirroring devices, the NIDS is typically configured with no IP address in order to protect direct attack on the NIDS device itself. Another NIDS network interface connected to the management console is configured with an IP address in order to manage NIDS software. The management console and NIDS are usually setup on a local or internal network. Managing NIDS over the Internet is considered risky and prone to attack as communication between management console and NIDS could be eavesdrop, attack and compromise, which would be disastrous for the whole network.

Host Intrusion Detection System

HIDS is a type of IDS software system that analyses the events occurring on a computer systems rather than network for detecting malicious events. HIDS closely examines the state of a computer system including hardisk activity, Random Access Memory (RAM) contents, OS processes and log files, and raises an alarm if it encounters any unusual or unexpected activity. For example, if a user with Read-only file access rights tries to modify the files content then HIDS will raise an alarm and notifies the administrators about ongoing activity. Figure 2.18 shows the event analysis flow in HIDS.

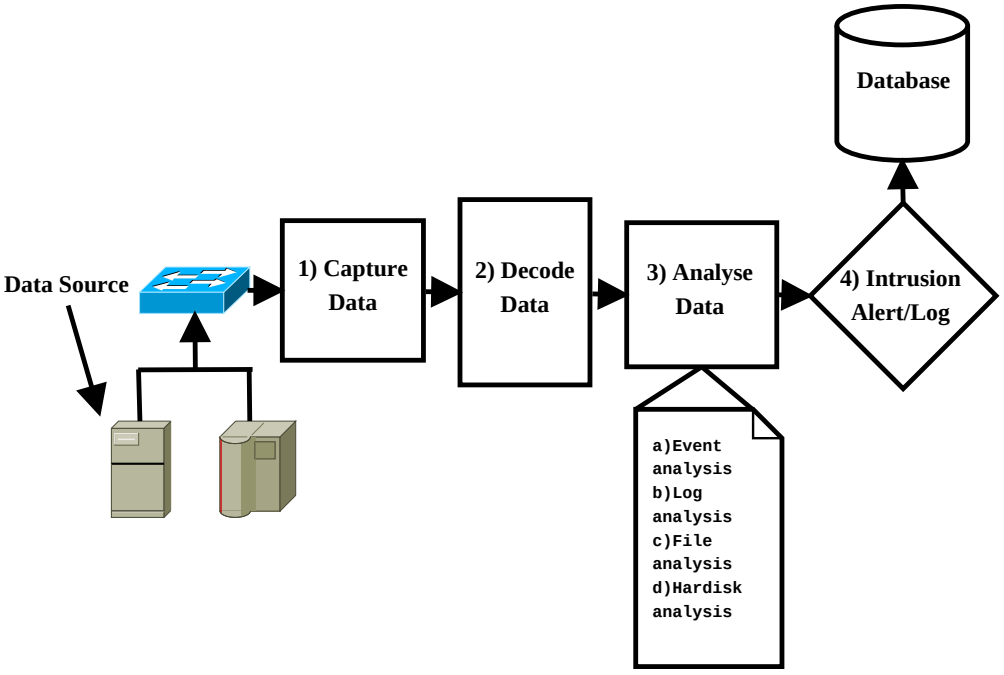


FIGURE 2.18: Typical HIDS data analyses flow

OS events are first intercepted by the HIDS's event capture component. These events are then decoded and the decoded data is stored to HIDS application memory for analysis. The data analysis component examines events data for any violation of system access rights or any similar suspicious activities. Finally, an intrusion alert component raises the alert and/or logs the events.

HIDS Deployment: HIDS normally install on a monitoring devices rather than on a crucial network points. HIDS is indeed a requirement for a system hosting private data and internet facing servers such as web servers, database servers. Industry standard for most intrusion detection systems mandate the use of both NIDS and HIDS. Figure 2.19 shows this kind of deployment.

HIDS agent software is typically installed on any LAN servers facing the Internet and providing services to external machines. HIDS agent software collects server machine

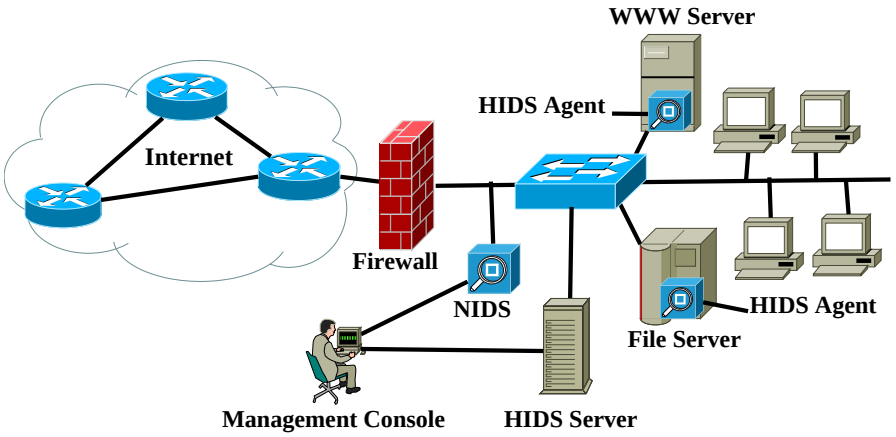


FIGURE 2.19: NIDS sitting between LAN and the Internet and HIDS agents on Internet facing servers

events and pass them on to the HIDS server software for event analysis. HIDS server has necessary information (e.g. system policies, rules, signature) in order to carry out analysis and if malicious activity is found it logs events or/and raise alarm(s) for the attention of Network Administrator. A management console is also usually connected to the HIDS server that keeps HIDS server information up-to-date with new policies and attacks.

2.4.4 Intrusion Detection Techniques

There are two approaches used by IDS to monitor events for the detection of malicious activity. *Signature detection* and *Anomaly detection*. Signature detection is the prevalent approach in IDS implementation, and is deployed in notable commercially successful products such as *TippingPoint IDS* and *Cisco IPS* series. Table 2.8 shows the summary of both detection techniques.

TABLE 2.8: Summary of IDS detection techniques

Technique	Method	Advantage	Disadvantage
Signature detection	Analyse data using database of attack signatures/patterns.	Accurate detection of attacks.	Unable to detect previously unseen attacks.
Anomaly detection	Analyse data using statistical techniques.	Detect previously unseen as well as old attacks.	Raise lots of false alarms.

Signature Detection

Signature detection techniques involve matching sets of known attack signatures with events occurring on a computer and network for any signs of malicious activity. Each

attack signature represents a known security threat (e.g. Virus, Worms, Spyware, DoS attacks etc).

An IDS that analyses data using signature detection techniques is known as *Signature based IDS (SB-IDS)*. A SB-IDS maintains a database of signatures used to checking the flow of network traffic or computer system events for the presence of malicious/suspicious activity. When any packet or computer events matches a signature an appropriate action is taken, which usually involve raising alarm(s), logging events and sending alert to the Network Administrator. A NIDS that uses signatures for detecting attacks in network traffic is called Signature based NIDS (SB-NIDS). Snort and TippingPoint are SB-NIDS with a feature to detect malicious events occurring on a network. OSSEC is a *Signature based HIDS (SB-HIDS)* that use signatures to analyses computer system events.

Anomaly Detection

Anomaly detection techniques use statistical methods such as frequency, variance, mean and standard deviation to analyse events and define if a pattern is normal or anamolous. In order to analyse these events, first a baseline of normal profile of users, networks, servers and workstations, server and application programs is created to detect anomalous events [33].

An IDS that uses anomaly detection to detect attacks is called *Anomaly based IDS (AB-IDS)*. Bro is an open source, Unix based NIDS that analyse network traffic using signatures as well as anomaly detection techniques [34]. Bro is primarily a research platform for intrusion detection.

Commercial IDS solutions prefer signature detection implementation over anomaly detection because it is difficult to define normal system and network profiles. Further, it becomes highly subjective as to what is normal, and what is an anomaly that could be an indicator of suspicious/malicious activity or attacks. Commercial anomaly detection products do not appear to be successful and none are evident. However, the main drawback of signature detection technique it can not let the IDS detect attacks that are previously unseen.

2.4.5 Popular Intrusion Detection System Products

The concept of intrusion detection for internet security has been around for nearly thirty years which was born with James Anderson seminal paper entitled, *Computer Security Threat Monitoring and Surveillance* [35]. Even after 30 years, only a small number of successful commercial and open source IDS products/applications are available. The

reason is very simple, an IDS is a highly sophisticated software system that must anticipate and detect malicious activity by analysing millions of generated events/packets. It is difficult for software vendors to successfully bring to market an effective IDS solution with the characteristics of accurately detecting malicious behaviour and intrusion. Current commercial and open source developers are unable to saturate the market of IDS technology and only a small number of vendors and developers are providing IDS solutions.

Table 2.9 and Table 2.10 shows the product details of leading HIDS and NIDS and private companies NIDS.

Table 2.9: Summary: Product details of leading HIDS and NIDS

Product	Release Date	Platform	Method	Description
Tripwire HIDS [36]	1992	Unix	Monitor file changes (File integrity checker)	Commercial products: Tripwire Enterprise and Tripwire for Server. Open source version moderated on Sourceforge.net
OSSEC HIDS [37]	2005	Linux, Solaris, Windows, MAC OS X, OpenBSD, FreeBSD	Signature and Anomaly detection	Owner (Trend Micro): Free and open source software
Bro HIDS [38]	1998	Unix	Signature and Anomaly detection	Open source NIDS for research purpose
Snort NIDS [3]	1998	Linux, Solaris, Windows, IBM AIX	Signature detection and Protocol analysis	Commercial product: Sourcefire IDS/IPS. Open source version available from snort.org

Table 2.10: Summary: Best NIDS/NIPS product of leading private companies

Company	Best Product	Method	Throughput
Cisco	IPS 4200 Model: 4270 [39]	Signature detection	4.0 Gbps
IBM	Proventia NIDS Model: GX6116 [40]	Signature detection	6.0 Gbps
Sourcefire	IDS/IPS Model: 3D9900 [41]	Signature detection	10.0 Gbps
TippingPoint	NIPS Model: 5100N [42]	Signature detection	5.0 Gbps

2.4.6 Issues and Limitations of Intrusion Detection System

IDS security technology is not a perfect security solution and limitations and weakness include sophisticated attacks, false alarms, zero-day exploits and speed of analysis (Sections 2.4.6). Complementary security tools and products include:- *vulnerability* scanners that probe ports of network servers for detecting anything malicious that could allow

unauthorised access, *File integrity checkers* to monitor important systems and private data for signs of unauthorised modifications, and *Honeypots* that are deployed as a trap machine with fabricated information designed to appear valuable in order to lure the hackers away from critical systems [43–45].

IDS limitations and weaknesses are now discussed.

Sophisticated Attacks

NIDS provide no mechanisms to deal with sophisticated network attacks such as DDoS which carries out with the help of botnets. An IDS cannot detect the location of botnet client or master machine nodes.

False Alarms

A false alarm in IDS is defined as, any ongoing network and computer activity that IDS seems suspicious and so raise alarm which actually neither malicious nor misuse of resources.

SB-IDS's have the potential of very low false alarm rate as they have exact signatures from an attack and malicious activity database which they use to compare network traffic and events occurring on a computer. In other words any action that is not explicitly recognised as an attack is considered acceptable. However, AB-IDS produce higher number of false alarms as these IDS have a baseline definition of normal event and activity that they use to distinguish between normal and anomalous events. The baseline is an estimated calculation of what is known as normal activity on a network and computers, which are effective to detect previously unseen attacks but raises a high number of false alarms because anything that does not correspond to a previously learned behaviour is considered intrusive.

False alarms are a fundamental issue for anomaly detection techniques and improvements in lowering the false alarm rate is an active area of research. Statistical and artificial intelligence (Artificial neural network) are the main methods used to create a baseline profile of normal network and computer usage for anomaly based IDS in order to detect anomalous activities [46].

Zero-day Exploits

A Zero-day exploit is defined as, a piece of malicious code used by hackers that exploits a software vulnerability unknown to the owner/user of software or developer or any other person.

SB-IDS's provide no ways to deal with zero-day exploits and other attack variants as their detection success relies on the provision of attack signatures. By definition a signature database has no definition of a zero-day exploit and the attack will be missed. AB-IDS can potentially detect zero-day exploits if they result in system/network behaviour that is sufficiently unusual but there is no guarantee of detection everytime and AB-IDS additionally suffers with the issue of false-alarms.

Speed Of Packet Analysis

The evolution of new hardware and network technologies has resulted in dramatic increases in network transmission link speeds. The contant increase in link speed requires high computational performance from NIDS as it must accomplish all processing in even less time. Possible solutions to counter this issue is to either reduce the overall data rate of the network, or simply drop some packets without analysis to keep up with the pace or rate of incoming/outgoing packets. Both solutions have drawbacks as one lower the network data rates or slows down the network and the other puts the network at increased security risk as some attack signatures can be missed to detect in drop packets.

Researchers came up with a different idea to tackle this ongoing problem. They identified the need to improve NIDS packet analysis speed using high performance processing hardware or a platform that is more suitable for packet processing. These platforms have been used to optimise algorithms to increase packet analysis speeds for NIDS. Commercial answer to this problem is also high performance processing platforms where NIDS sold as standalone packet analysis devices that are ready to be deployed in a network. Successful commercial products such as Cisco NIDS, Tipping NIDS, Sourcefire NIDS and IBM all selling these standalone devices.

2.4.7 NIDS Computationally Demanding Process

There are three main computationally demanding process of SB-NIDS that are a performance bottleneck on high data rate network and slow down SB-NIDS packet analysis speed: *Stateful Packet Inspection (SPI)*, *Packet Classification* and *Pattern Matching*.

Figure 2.20 shows the typical internal architecture of an SB-NIDS to demonstrate the packet analysis flow depicting the three main computationally demanding processes.

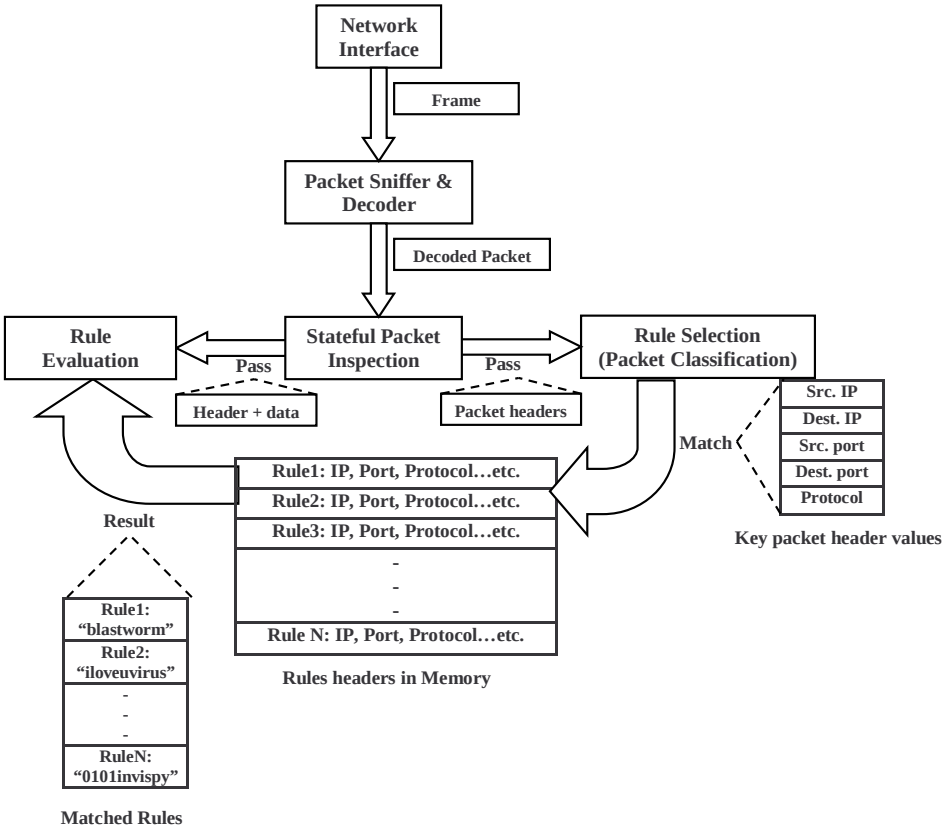


FIGURE 2.20: Packet Inspection in SB-NIDS

An SB-NIDS Packet Sniffer component first captures a network packet from a network interface and uses the Packet Decoder to decode the packet protocols. The decoded packet protocol data is then store in a SB-NIDS application memory area. This decoded packet data is then analyses in series of stages. In the case of stateful protocols such as TCP, the SPI component uses the key packet header values (IP addresses and Port numbers) to check the packet association with one of the established TCP connections. Next the packet higher layer protocol data is analyse using SB-NIDS dedicated software components that checks validity of protocol by looking for invalid protocol values that may lead to protocol related attacks. Finally, packet header and payload content is searched for invalid header values and malicious patterns with the help of Rule selection and Rule evaluation component.

Stateful Packet Inspection (SPI)

Stateful Packet Inspection (SPI) is a process to track each and every network connection traversing through the network in order to look for any illegal connection trying to attack

the network services and equipments. SPI helps to detect different network attacks such as TCP SYN DoS attack and port scanning/probing.

For each TCP connection successfully established, the SPI module assigns a unique ID number by computing a hash value on key packet header values typically IP addresses and port numbers. The unique ID number is then store in a connection state table in main memory. For every incoming packet, the SPI module computes the hash value using IP address and port number and looks up the connection state table in order to check that the packet belongs to one of the established connections. SPI is a bottleneck on a high data rate network due to frequent memory access that directly effects packet analysis speed. Also key packet header values are also hash in SPI which also degrades packet analysis speed performance. Javier et al carried out a study on SPI to measure its impact on packet processing speed under high volume traffic [47, 48]. They observed the packet analysis speed was significantly degraded with an increasing number of active connections.

Packet Classification

Packet classification is the process of selecting attack signatures or rules for comparison with packet data in order to detect malicious activity. These rules are selected by matching key packet header values (IP addresses, Port numbers and Protocol type) with those specified in a part of rule known as *Rule header*. In the context of SB-NIDS, packet classification can also be referred as *Rule Selection*.

Rule selection involves matching source and destination IP addresses and ports, and protocol types. Protocol type is a simple matching process which involve only constant values, whereas IP addresses and ports occasionally involve matching a range of values which is complex and computationally demanding. This process becoming more complicated due to regular increases in both network data rate and the number of attack rules. Lakshman, and Stiliadis, Gupta and McKeown and Srinivasan et al developed a novel algorithm to perform fast packet classification [49–51]. Mcauley and Francis and Florin et al optimised the packet classification processing speed performance using high performance processing platform [52, 53].

Pattern Matching

Pattern matching in SB-NIDS involve comparing packet payload contents with known malicious patterns specified in part of rule known as *Rule options*. Pattern matching

is carried out using exact pattern matching algorithms. These pattern matching algorithms execute slowly on general purpose processors typically requiring a high number of memory accesses and comparison operations. For example, a C-programming language method *strcmp(char *str1, char *str2)* compare characters requires one memory access to fetch each character into a CPU register and a comparison per character. On a device such as an FPGA, an optimised string comparison functional unit can be developed to check multiple characters in one clock cycle.

Notable pattern matching algorithms include Boyer-Moore and Aho-Corasick [54, 55]. These algorithms are still used in many network and system softwares such as Snort and grep, and are also currently the subject of pattern matching optimisation research for network applications (Section 3.5).

2.5 Summary

This chapter began by looking at a range of network security issues concerning two core problems: i) flawed design of network protocols in particular Internet Protocol suite (TCP/IP) and ii) the vulnerabilities in software applications and OS. Some other network security threats such as malwares also discussed to understand and quantify the scale of network security problem. To minimise and counter these network threats and stop the spread of malwares some core network defence mechanisms are discussed. To illustrate the effectiveness and limitations of network security technology, some defence mechanisms are discussed with a main focus on detail explanation of Intrusion Detection System (IDS), the main issues concerning IDS technology, its limitations and performance are focussed on Signature-Based Network Intrusion Detection System (SB-NIDS).

Chapter 3

Survey and Related Work

Monitoring everyday network traffic for an attempted intrusions and complex kinds of network attack is not a simple task for Network Administrators [56]. This task is increasingly difficult due to the huge volume of legitimate network traffic and constantly increasing network data transfer rates. Simultaneously, hundreds of software based solutions for network monitoring, debugging, surveillance and intrusion detection has been developed [57]. Some notable software solutions are NMAP, Netcat, Metasploit for network monitoring and debugging [58–60]. Other solutions are Snort, Bro, Cisco NIDS for detecting network intrusions and attacks, commonly known as Intrusion Detection System (IDS) or Network Intrusion Detection System (NIDS) [3, 38, 61] (Section 2.4). IDS or NIDS, particularly Signature based Network Intrusion Detection System (SB-NIDS) collect and analyses network traffic in real time by capturing packets directly from network interface(s). On high speed transmission network with data rate of gigabit per seconds or over, SB-NIDS struggles to perform packet analysis of every incoming or outgoing network packet. Consequently, SB-NIDS data buffer becomes full which force SB-NIDS to remove or drop some packets from packet buffer. This happens due to complex process of data collection, manipulation and analysis of network data in NIDS components (Section 2.4.7).

3.1 Chapter Roadmap

The rest of the chapter is outlined as follows:

- In section 3.2, introduction of how state of the art SB-NIDS is explained in the rest of this chapter. The basic technique and compare and contrast are followed in state of the art discussion.

- In section 3.3 the state of the art are explained. The main focus is on describing the state of the art SB-NIDS and pattern matching implementations. State of the art are grouped into categories and their contributions are clearly explained.

3.2 Introduction to Literature Review

The three computationally demanding process of SB-NIDS has been the subject of research and development. They can be distinguished as three independent areas of research (Section 2.4.7). The contribution of research is the state of the art high speed algorithms and hardware architectures that are capable of supporting high data rate throughput. In this survey, state of the art of only one area *(Pattern Matching)* is discussed in detail due to its direct association with part of the research presented in this thesis (Section 3.5). Additionally, some state of the art are also discussed that were proposed to optimise the SB-NIDS using high performance processing and computing technology to support packet analysis at high data rate network link. Figure 3.1 and figure 3.2 shows the state of the art related work.

3.3 Literature Explanation

First the state of the art SB-NIDS design and implementation is discussed which are distinguished in categories and compared (Section 3.4). This is followed by the discussion of state of the art pattern matching algorithms and hardware architectures (Section 3.5). Different pattern matching implementations is identified into categories and compared.

3.4 SB-NIDS using High Performance Computing Platform

It has been observed that hardware architectural approach is used consistently to optimise the packet analysis speed performance of complete SB-NIDS to enable high data rate network traffic analysis. The two hardware architectural approaches for optimising the complete SB-NIDS are: *Computer Clusters* and *Embedded Processing Platform.*

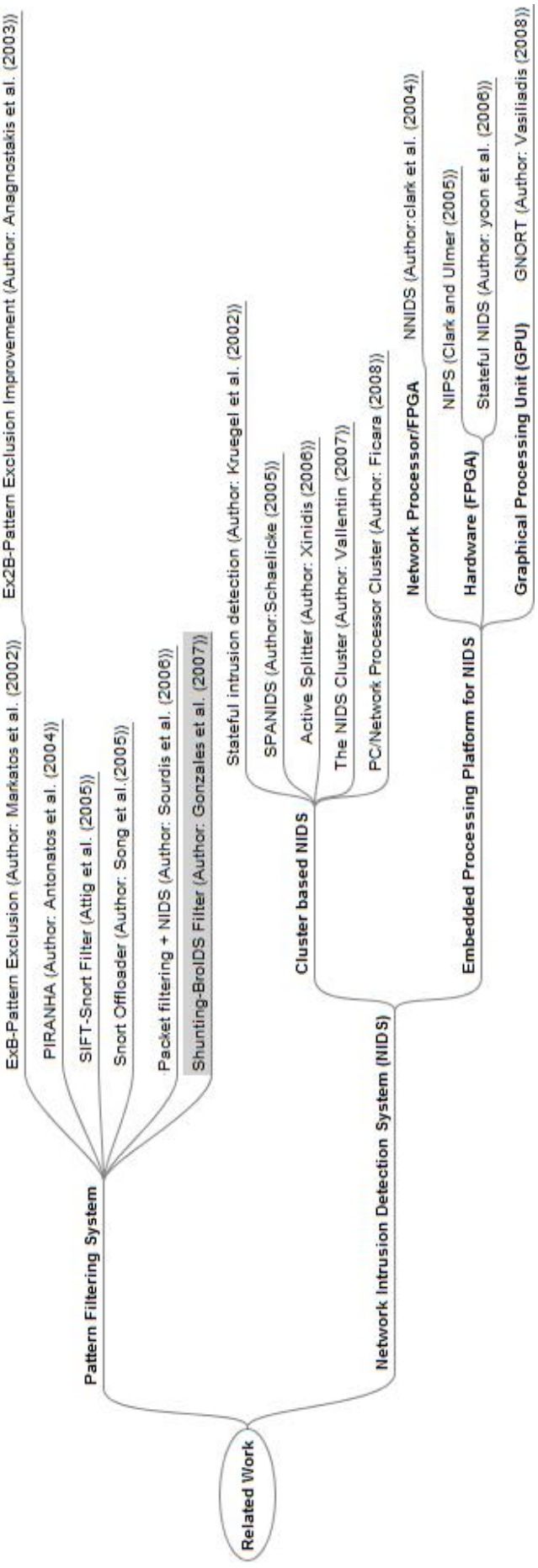


FIGURE 3.1: Network Intrusion Detection Systems and Filtering Systems

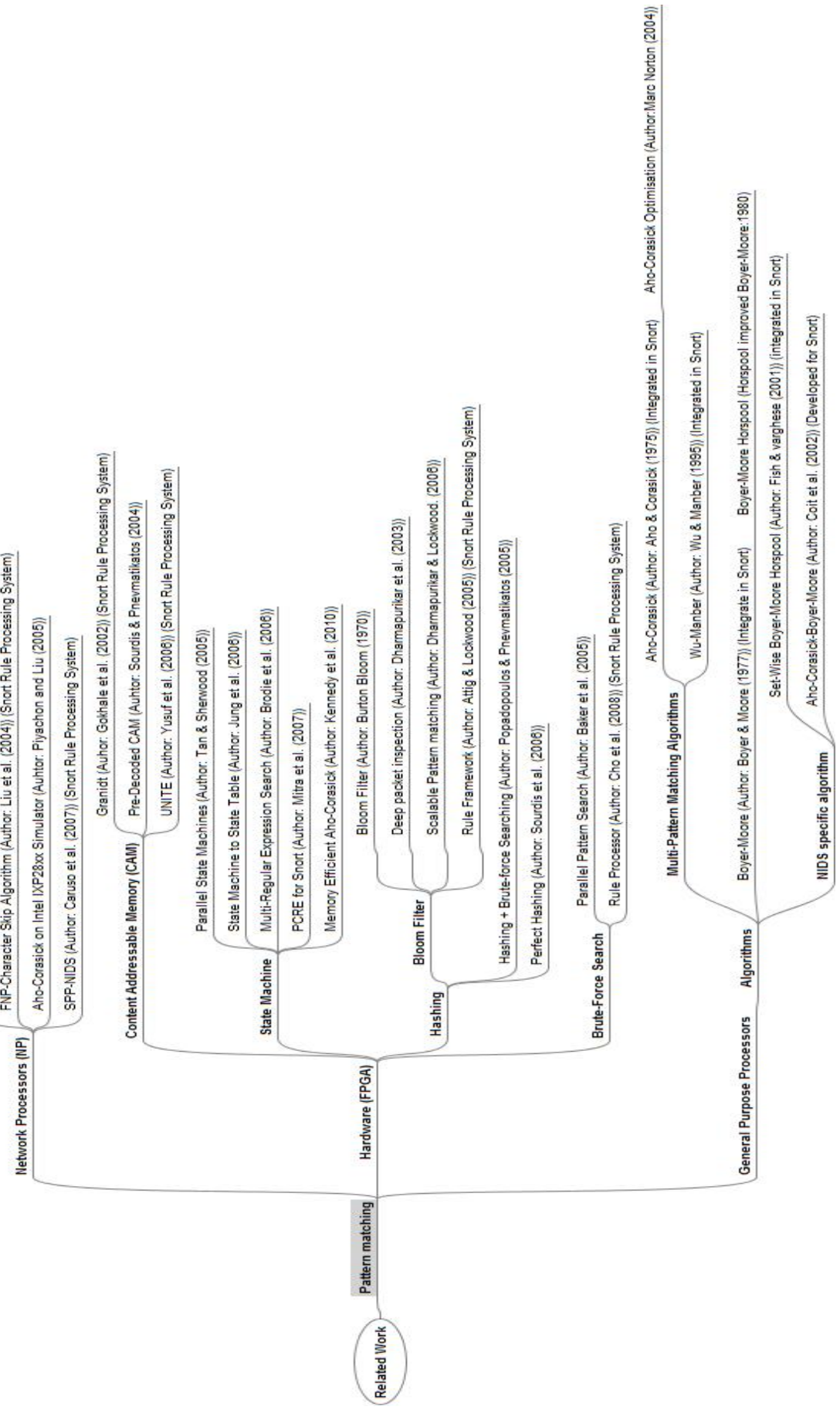


FIGURE 3.2: Pattern Matching

TABLE 3.1: Summary of Computer cluster and Embedded processing based SB-NIDS

Hardware Architecture	Authors	Summary
Computer Cluster	Kruegel et al [62]	One of the earliest ideas that employed cluster of general purpose processor (PC) for deploying Snort for stateful and distributed packet analysis.
	Schaelicke et al [63]	A Loadbalancer design that supports dynamic feedback mechanism to ensures dynamic adjustment of network traffic distribution in order to avoid particular NIDS sensors overloaded with too much traffic.
	Xinidis et al [64]	A sophisticated Loadbalancer that perform packet distribution as well as packet filtering, locality buffering and TCP packet reassembly.
	Vallentin et al [65]	Hardware architecture with cluster of general purpose processors (PC) for the deployment of SB-NIDS and has mechanism to detect and recover from NIDS node failure.
	Ficara et al [66]	A cluster based NIDS architecture where the SB-NIDS deployed on a cluster of general purpose processor (PC) and connected via Loadbalancer implemented on Network processor performing packet distribution as well as packet filtering and packet re-ordering.
Embedded Processing	Clark et al [67]	Network Node Intrusion Detection System (NNIDS) for packet analysis at host (PC) level developed by porting Snort components on a Network Processor.
	Clark and Ulmer [68]	A Network Intrusion Prevention System (NIPS) implemented on FPGA that perform packet analysis by monitoring multiple Gigabit Ethernet links.
	Yoon et al [69]	FPGA based security system with management subsystem for updating security policies and analysis subsystem for network packet analysis that performs stateful packet inspection and signature checking.
	Vasiliadis et al [70]	A modified Snort where Packet capture, Decoder, Preprocessor and Logging Engine executes on CPU and Snort detection engine is offloaded to Graphic Processing Unit (GPU) for rule evaluation.

3.4.1 Computer Clusters for SB-NIDS

Such SB-NIDS deployed with cluster of NIDS packet analysis engine or sensors that analyses fraction of distributed network traffic. Figure 3.3 shows the typical arrangement of hardware for cluster-based SB-NIDS.

The core hardware in such architecture is a Load balancer which distributes the network traffic for analysis by cluster of SB-NIDS. Load balancer can either be implemented with

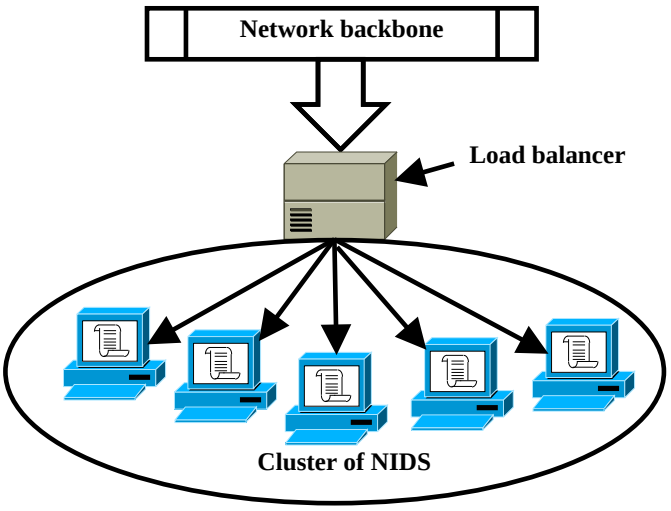


FIGURE 3.3: Typical arrangement of hardware for Cluster-based SB-NIDS

embedded processing hardware technology or a standalone PC with a Load balancing software. A SB-NIDS cluster can be made up either by installing NIDS software package on standalone PCs or by deploying high performance SB-NIDS packet analysis processing technology commonly available in commercial market (Section 2.4.5).

The main reason for such a complex arrangement of hardware (Table 3.2) is to perform network packet analysis on a very high throughput preferably gigabit per second rate of network throughput. One of the important components that perform critical function in attaining such a high speed packet analysis in NIDS cluster is the *Load balancer*. The main function of the Load balancer is to distribute the network packets to the SB-NIDS cluster. Such a distribution should be fair in terms of packet distribution and load, and should also maintain the state of the network connections or flows which is critical for detecting network attacks. Distributing network traffic in such a manner is not at all simple. Therefore, the proposed SB-NIDS cluster systems main emphasises was on strategies and techniques of equal, efficient and stateful distribution of network traffic between SB-NIDS clusters [62–66].

Research contribution

One of the first ideas of distributed packet analysis using computer clusters proposed in 2002 by Kruegel et al. [62]. Their distributed NIDS architecture employed the cluster of NIDS engines or *sensors* install on ordinary PC with general purpose processor. Each NIDS sensor is responsible for the detection of specific attacks due to careful distribution of attack rules between clusters of sensors in a manner that each sensor searches the packet only for particular type of network threat or attack.

Table 3.2: NIDS cluster hardware specification

Authors	Hardware Description
Kruegel et al [62]	Scatterer: (Intel Xeon 1.7 GHz Processor). Traffic Slicer and Reassembler: (Intel Pentium IV 1.5 GHz). Network Switch: (Cisco Catalyst 3500XL).
Schaelicke et al [63]	Simulation Environment on general purpose processor: Hardware specification not specified.
Xinidis et al [64]	Splitter prototyped on: (Radisys ENP 2506 board with Intel IXP1200 Network Processor: One ARM processor and six special purpose processor (Microengines)). NIDS nodes: (Dell PowerEdge 500SC with Intel Pentium III 1.13GHz) and (Dell PowerEdge 1600SC with Intel Pentium IV Xeon Processor 2.66 GHz).
Vallentin et al [65]	Frontend and Backend nodes: (Intel Pentium D 3.6 GHz dual-CPU) at LBNL. Frontend node: (Dell PowerEdge 850 with Intel Pentium D 920 Dual-core) and Backend node: (Sun Fire X2100 with AMD Opteron 180 Dual-core) at UC Berkeley.
Ficara et al [66]	Hardware Classifier: (Radisys ENP-2611 board with Intel IXP2400 Network Processor. Intel Xscale 32 bit RISC process and eight special purpose processor (Microengine)). NIDS node: (Intel Xeon 2.8 GHz Processor).

Table 3.3: Advantages and disadvantages of cluster based NIDS

Authors	Advantages	Disadvantages
Kruegel et al [62]	Considered first proposed NIDS using computer cluster with Stateful loadbalancing.	Expensive set of Loadbalacing hardware
Schaelicke et al [63]	Dynamic loadbalancing feedback mechanism for equal load distribution.	Stateless loadbalancing.
Xinidis et al [64]	An active loadbalancer (Splitter) that also filter traffic as well as process some NIDS function.	Packet distribution based on rules groups provide a way to attack and fail the system by overloading specific NIDS node with packets.
Vallentin et al [65]	Spare hardware for on the fly replacement of failed NIDS sensor.	No fail recovery mechanism available for loadbalancer hardware failure.
Ficara et al [66]	Loadbalancer (Classifier) performs NIDS packet classification process and offload some NIDS processing load.	Unsupportive to stateful and protocol analysis because classifier hardware forward only those packets that match rule headers.

Their NIDS cluster architecture is comprises of an array of complex sets of hardware. The packet distribution occurs with the help of these hardwares. In this architecture, first the *Scatterer hardware* captures the network packet from the network interface and forward them to the sets of *Traffic Slicer hardware*. Traffic slicers then further distributes these packets to the appropriate *NIDS sensor* based on Snort on a general purpose processors. These packets travels through the arrays of *Reassembler hardware* which are indirectly connected to Slicer hardware via Network switch and directly connected to NIDS sensors. Before packets finally passes on to the NIDS sensors for analysis, the Reassemblers

arranges the packet order on the basis of first-capture-first-forward basis. The main idea of the slicing mechanism is to distribute the packets to multiple NIDS sensors in order to gain packet analysis efficiency and support of higher rate throughput. This architecture also supports SPI and can analyse packets at over 190 Mbps on each sensor. One of the best features of this cluster based NIDS is scalability which can be achieved by adding easily an extra NIDS sensors. The main disadvantage of this complex but highly distributed cluster based NIDS are high first time investment and maintenance cost. There is also an operational issue which is SB-NIDS sensors dependency on one and only centralised packet Scatterer hardware that can make the complete SB-NIDS system non-functional in case of scatterer hardware failure. Furthermore, this distributed SB-NIDS architecture provides no *Dynamic feedback mechanism*, a mechanism that dynamically adjust network traffic distribution due to network traffic flow change in order to avoid particular SB-NIDS sensors overloaded with too much traffic[1]. In summary, this system is a scalable SB-NIDS solution but requires high first time investment and maintenance cost and also lack Dynamic feedback mechanism feature that is essential to take control on uneven packet distribution flow.

In 2005, Schaelicke et al came up with a design of efficient Load balancer hardware on FPGA for the cluster based SB-NIDS approach called *SPANIDS* [63]. This Load balancer hardware has a gigabit Ethernet interface to capture the network packet and distribute them in a stateless manner between the cluster of NIDS sensors using the flow based network traffic distribution approach. When Load balancer captures the packet, it extracts the IP addresses and port numbers from the network packet for flow based distribution. The Load balancer then hashes these values into a table created in the Load balancer local memory or RAM. Each table entry is associated with specific NIDS sensor responsible for analysis of particular flow of packets. Figure 3.4 shows this process.

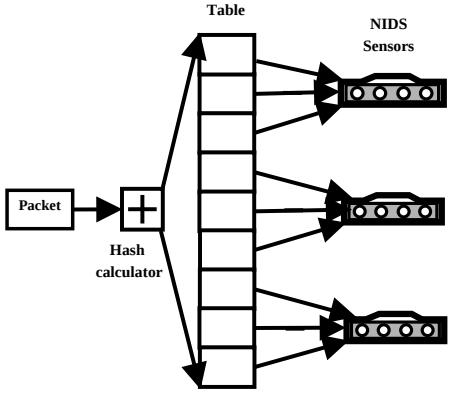


FIGURE 3.4: Loadbalancing using hash calculator

[1]flow based network traffic distribute traffic based on IP address and/or port number.

Network traffic distribution using this kind of hashing or flow based distribution can easily overload any SB-NIDS sensor in cluster. In order to overcome this issue dynamic feedback mechanism is incorporated into the design of this efficient Load balancer which is the main contribution of their work not supported by Kruegel et al. [62]. The dynamic feedback mechanism is supported with the help of simple communication protocol implementation. NIDS sensors communicates with the Load balancer with flow control message to notify the traffic load on the SB-NIDS sensor. The Load balancer then adjusts the traffic on the loaded SB-NIDS sensor by diverting some of the flows to the least loaded SB-NIDS sensor. Such an adjustment in response to a flow control message disturbs the flow based analysis due to the movement to different sensors. The major disadvantage of this proposed Load balancer is the heavy reliance on one Load balancer hardware that in case of failure makes the complete SB-NIDS cluster non-functional, the same problem shared by the SB-NIDS cluster design proposed by Kruegel et al. [62]. The simulation model of the cluster based system is created to evaluate the dynamic feedback mechanism performance. It consist of 12 simulated sensors, each with packet buffers of the Linux default size of 64 Kbytes and a Load balancer FPGA hardware. The experiment conducted with the 21 hour network traffic trace shows that the dynamic feedback mechanism is able to drastically improve the number of packets drop. Without feedback mechanism, a total of 498,995 packets are dropped, while feedback mechanism reduces the total to 46,208 packets drop which is the significant improvement. In summary, the efficient Load balanacer hardware design implemented in FPGA is presented which has a advantage over Kruegel et al load balancing approach due to dynamic feedback mechanism features but both design share the same weakness of single centralised Load balancer hardware with no support of failed recovery mechanism [62].

In 2006, Xinidis et al proposed the concept of active Load balancer for cluster based SB-NIDS [64]. Unlike passive Load balancer that distributes only the network traffic between NIDS nodes; active Load balancer proposed in this design performs not only the packet distribution but also supports useful features to improve the packet analysis speed. This include *Packet filtering* (Section 3.5.2), *Locality buffering* and SB-NIDS processing such as *TCP packet reassembly*. Packet filtering involve the processing of header only attack rules then forwarding rest towards the SB-NIDS sensor via the locality buffer. Locality buffering is a technique that is applied by reordering the network packet using locality buffer of SB-NIDS sensor. This result in improve packet analysis speed performance due to reduction in cache misses. The packet reordering criteria is crucial for successful implementation of locality buffering which is actually arrangement of stream of packets in a way that each SB-NIDS sensor trigger the same set of attack rules everytime it received network packet[2]. This active Load balancer is prototyped on Intel IXP1200 Network

[2]In Snort, rules are arranged based on matching headers forming rule-groups.

processor which performs flow based packet distribution by hashing key packet header values (IP addresses and port numbers). Cluster of SB-NIDS sensors based on Snort (ver 2.0 and 2.0.2) with active Load balancer executed on PC with general purpose processor and is tested with traces of network packets collected in September 2002 on NLANR network shows the significant performance improvement in terms of reducing overall the processing load on NIDS sensors. The test results demonstrated reduction of overall 8 % traffic by filtering and 10-17 % by locality buffering. The Load balancer throughput measured for 64 bytes packet is 500 Mbps and for 1472 bytes packet is 980 Mbps. In summary, the first ever active Load balanacer hardware design is presented on network processor which performs packet distribution and actively engaged in other processing to reduce the processing load from cluster of NIDS sensors. The overall capability of the Load balancer is increased which is the novel contribution of this work but lacks in features like dynamic feedback mechanism for flow control and failed recovery mechanism necessary to coup with failure of one and only centralised active Load balancer.

In 2007, Vallentin et al proposed a design of cluster-based SB-NIDS architecture using the combination of Frontend node and Backend node hardware [65]. SB-NIDS sensors are represented as *Backend nodes* is the cluster of PCs with general purpose processors. The Load balancer is represented as Frontend node has a gigabit Ethernet interface to capture the network packet for distribution to SB-NIDS sensors for packet analysis. This distribution is flow based which is performed by extracting IP addresses and port numbers from packet and hashes them to generate hash value. This hash value is then taken the modulus with the total number of SB-NIDS sensors for selecting the packet destination sensor for packet analysis. Once the destination sensor identify, the Load balancer writes the destination NIDS sensor MAC address and forwards the packet. This cluster based system is very similar to other previously discussed cluster based NIDS with the exception of failed recovery mechanism to enable recovery of SB-NIDS sensor in case of failure. To perform this task an additional hardware called *Hot Spare* hardware is installed in a system which monitors the NIDS sensor via a ping like method known as heartbeat mechanism. If any SB-NIDS sensor found failed then the Hot spare takes the charge of monitoring all the traffic flows destined to failed SB-NIDS sensor MAC address. The single and most common issue is the centralised Load balancer with no failed recovery mechanism for Load balancer. The most promising characteristic of this system is the packet inspection support of 10 Gbps data rate throughput with the FPGA based Load balancer. In summary, a cluster-based SB-NIDS is presented with an added feature of failed recovery mechanism for SB-NIDS sensor which advances the state of the art.

In 2008, Ficara et al proposed a cluster-based SB-NIDS architecture which comprise of NIDS sensors deployed on cluster of PCs on general purpose processor and Load balancer

on Network Processor [66]. SB-NIDS sensors and Load balancer are connected directly with gigabit Ethernet link. Load balancer is also connected with a network backbone. Once it receives the packet, it then extracts key packet header values (IP addresses and ports) and hashes them to generate hash values. This hash value is compared with the pre-computed hash values of attack rule headers group stored in Network Processor local memory [71]. If any rule group header hash values match with key packet header hash values then the packet is forwarded to the NIDS sensor responsible to evaluate the particular rule group on packet. In summary, this cluster-based SB-NIDS is a lower cost PC based SB-NIDS; this SB-NIDS has no distinguished features like failed recovery mechanism or dynamic feedback mechanism.

In summary, cluster-based SB-NIDS are efficient and provide high throughput SB-NIDS packet analysis. Majority of cluster-based SB-NIDS requires high cost hardware investment and future maintenance.

3.4.2 Embedded Processing Platform for NIDS

High performance embedded processing and computing platform has also been used to implement optimised SB-NIDS solutions. Most of these commercial solutions of SB-NIDS is sold as a "Box solution" in a commercial market that are implemented with embedded processing platform (Section 2.4.5).

Research Contribution

One of the earliest work that demonstrates the design and implementation of SB-NIDS using high performance embedded processing platform is proposed by Clark et al in 2004 [67]. This NIDS is called *Network Node Intrusion Detection System (NNIDS)*. It is a SB-NIDS implementation using Snort on Network Processor for distributed packet analysis. The unique idea of implementing the NNIDS on a network processor comes from their believe that the Network Processors can be easily integrated into a Network Interface Card (NIC) of any computer or node which will easily enable distributed packet analysis on network. NNIDS is developed on Radisys ENP-2505 development board. Main components of the board are Intel IXP1200 Network processor that has a StrongARM processing core and six microengines (Processors) with a clock speed of 232 MHz and Xilinx Virtex-1000 FPGA co-processor attached with IXP1200 processor via PCI mezzanine connector (PMC). Snort apart from pattern matching is ported on Intel IXP1200 network processor while the pattern matching is offloaded to FPGA co-processor for high speed pattern matching. The test result of NNIDS shows that the pattern matching component of the system is able analyse network traffic up to the

951 Mbps. NNIDS also only able to perform SPI but does not provide any facility to perform application level protocol (DNS, SMTP, FTP, HTTP etc.) analysis. In summary, it is first ever Snort port on Network Processor architecture with FPGA based pattern matching hardware acceleration unit that improves Snort slow speed search of attack signatures nearly up to 2 to 3 times when compared with Snort's pattern matching throughput on general purpose processor but still lower in comparison to other state of the art FPGA based Pattern Matching Hardware Accelerator (PMHA) which performs packet analysis up to 1.85 Gbps throughput (Section 5.3). Also this system propose the unique concept of enabling distributed packet analysis by integrating SB-NIDS in NIC and so advances the state of the art.

In 2005, Clark and Ulmer proposed a Signature-based Network Intrusion Prevention System (SB-NIPS) design that performs inline packet processing in order to stop and prevent network attacks [68]. It is implemented on a Xilinx Virtex II Pro (V2P7-6) FPGA device on a ML300 embedded development board. The unique feature of this system is the support of packet analysis on multiple gigabit Ethernet links using Snort's attack rules. These attack rules are translated into hardware configurations for the FPGA by implementing a program using JAVA Hardware Description Language (JHDL). This SB-NIPS has two main units, a *Network Interface* (NI) unit and a *Intrusion Detection* (ID) unit. Multiple NI units are connected to two different network links and supplies network packets to an ID unit for network threat detection and prevention. Due to the limited chip area of the V2P7 FPGA, this system was only tested with 21 rules and achieved a maximum throughput of 8 Gbps. Clark and Ulmer also implemented this design to a larger Xilinx XC2VP100 Virtex-II Pro FPGA chip with 1299 Snort attack rules (17514 characters) and utilised 36 % of the chip's LUTs and 47 % of its slices. This design is not very efficient in comparison to the FPGA based PMHA which is implemented with more than 3 times Snort attack rules (9140) and utilised 56 % of FPGA logic blocks and 66 % of slices of a Xilinx XC2V6000 Virtex-II FPGA (Section 5.3). Also the ID unit implementation of this SB-NIPS does not perform any protocol based analysis. In summary, it is the first SB-NIPS design with both packet capture and intrusion detection and prevention components implemented on FPGA. Not fast enough in comparison to PMHA presented in this thesis.

In 2006, Yoon et al proposed a NIPS architecture called *Next Generation Security System* (NGSS) [69]. NGSS is implemented on a Xilinx Virtex-II Pro XC2VP70 FPGA with Verilog Hardware Description Language (HDL). NGSS is made up of two systems: Security Gateway System (SGS) and Security Management System (SMS). SMS is just a management system that updates security policy (Configuration and rules update). SGS is the core network traffic analysis unit with three FPGA based hardware modules: Anomaly Traffic Inspection Engine (ATIE), Pre-Processing Engine (PPE) and Intrusion

Detection Engine (IDE) implemented on Xilinx Virtex-II Pro XC2VP70 FPGA. PPE module performs the SPI and maintains the session state table in CYNSE70256 9 Mbits TCAM and 2 MBytes Cypress SRAM. IDE module performs the packet classification for rule selection by comparing key packet header fields (Protocol, IP address and port number) with rule headers stored in TCAM memory. IDE module also performs the pattern searching in packet payload using on-chip FPGA memory in which Snort's attack patterns are stored. ATIE module generates alert messages and also performs intrusion prevention actions. NGSS is tested with only 200 Snort rules which is a reason why a system perform analysis with such a high throughput of 2.0 Gbps which is better than the PMHA throughput presented in this thesis but much lower in number of attack rules (Section 5.3). In fact this system provides high speed packet analysis but it does not provide functionality to perform different application level protocol analysis *(HTTP, DNS, FTP, Telnet, DNS etc.)*. This makes it susceptible to perform different kinds of protocol based attacks (Tiny fragment attack, DNS amplification attack etc). In summary, NGSS advances the state of the art by providing NIPS design that able to perform SPI as well as attack signatures checking at high throughput on a tightly coupled embedded processing platform but lacks with feature of protocol analysis. This feature is provided with a NIDS prototype on embedded processing platform presented in this thesis (Section 5.2).

In 2008, Vasiliadis et al demonstrated the use of Graphical Processing Unit (GPU) to speed up the packet analysis speed of SB-NIDS [70]. The SB-NIDS development on GPU is carried out using Snort and so this SB-NIDS is named as *Gnort*. Gnort implementation involved offloading Snort's computationally demanding packet processing operation (pattern matching) from a CPU to GPU. This is carried out by executing Snort's *Packet Capture*, *Packet Decoder*, *Preprocessor* and *Logging plug-in* components on a PC with general purpose processor and its only *Detection Engine* component which performs the most computationally demanding operation is offloaded for execution from a CPU to GPU. Packets when capture by the packet capture components on CPU first go through Packet Decoder and Preprocessor components and then pass on to the Detection Engine on GPU for attack signature search in packets. Packet transfer from CPU to GPU performed in bulk rather than every single packet. This is due to the overhead associated with every packet transfer. CPU use buffer to store packets before they transfer them to GPU which helped attaining higher data transfer throughput between the CPU and GPU. Also this transfer is supported by Direct Memory Access (DMA) feature. Once the packets received by GPU it is searched for the presence of attack signatures by Aho-Corasick (AC) multi-pattern matching algorithm executing on GPU stream processors. If any pattern found in a packet then GPU pass the attack signature detection

information back to CPU for alerting administrator and logging. Detection Engine component which performs pattern matching is implemented on NVIDIA GeForce 8600GT card which contains 32 stream processors arranged in 4 multiprocessors, operating at 1.2 GHz frequency and has 512 MB memory. Rest of the components implemented on Intel Pentium IV 3.40 GHz processor with 2 GB of memory. Test result shows that AC on GPU consistently achieved 1.40 Gbps compared to 600 Mbps on PC with general purpose processor. In summary, Gnort novel contribution is the use of implementation platform that provides complete SB-NIDS solution with high speed pattern matching on GPU. An integrated NIC on GPU would be a better option if ever available, to implement high speed network packet processing applications. Gnort pattern matching throughput is 1.40 Gbps which is lower in comparison to the FPGA based PMHA design presented in this thesis that supports 1.85 Gbps throughput (Section 5.3).

In summary, SB-NIDS design using embedded processing platforms is cost effective solution in comparison to cluster based SB-NIDS. Therefore, embedded processing platform is more appealing for developing and optimising SB-NIDS. Most state of the art apart from Gnort also struggle to provide features necessary to detect wide range of attacks in comparison to the solution presented in this thesis (Section 5.3). The pattern matching optimisation carried out using FPGA as part of the state of the art solution also does not support full attack rules due to limited FPGA resources in comparison to the PMHA presented in this thesis (Section 5.3).

3.5 Pattern Matching for SB-NIDS

Numerous research work has been done in past to optimise the pattern matching speed for SB-NIDS [72–79]. Some research work proposed novel pattern matching algorithms for NIDS to improve the packet analysis speed [72, 73]. Others research work came up with packet filtering technique in order to reduce the amount of traffic inspected by pattern matching algorithm of Snort [74–80]. Current research focus is exploring reconfigurable hardware [81–90] and network processors for pattern matching speed acceleration [91–93]. Almost all proposed pattern matching design used Snort attack rules for implementation and testing of their solutions. Out of these solutions, few proposed solutions to specifically optimised packet analysis speed of Snort. These solutions actually optimised the rule evaluation process of Snort which involved packet classification (Rule selection) and packet analysis (Packet header and payload check with rules). These solutions can be named as *Snort Rule evaluation system* which has been implemented with FPGA hardware, hybrid hardware/software platform and Network Processor for the improvement of packet analysis speed [85, 87, 90–92, 94].

3.5.1 SB-NIDS Specific Pattern Matching Algorithms

These algorithms were proposed specifically to optimise the pattern matching of Snort SB-NIDS. These algorithms came up with better results of search speed which is achieved by designing an algorithm that utilise the best features of state of the art pattern matching algorithms that include Boyer-Moore (BM), it's variant by Horspool and Aho-Corasick [54, 55, 95]. Due to this reason these two novel algorithms can be collectively called as *Hybrid Pattern Matching algorithms* [72, 73].

TABLE 3.4: NIDS specific hybrid multi-pattern matching algorithms

Authors	Hybrid Algorithm	Description
Fish and Varghese [72]	SBMH	Hybrid features adapted from Horspool (Boyer-Moore (BM) variant) and Aho-Corasick(AC) algorithms.
Coit et al [73]	AC_BM	Combined features in one algorithm adapted from Boyer-Moore (BM) and Aho-Corasick (AC) algorithms.

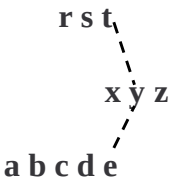
Research Contribution

In 2001, Fish and Varghese proposed the first hybrid SB-NIDS specific pattern matching algorithm called *Set-wise Boyer-Moore-Horspool (SBMH)* [72]. This hybrid algorithm adapts the Horspool variant of the Boyer-Moore algorithm and Aho-Corasick algorithm. This hybrid algorithm simultaneously match a set of patterns in just a single iteration or loop by applying a multi-pattern matching search technique of Aho-Corasick algorithm[3]. Also it adapt the Horspool bad-character heuristic search technique to skip character during pattern comparison to optimise further the pattern matching[4]. The multi-pattern search is applied by creating a suffix pattern tree and Horspool bad-character heuristic is applied by creating a bad-character shift table. Both the suffix tree and bad-character shift table is created by pre-processing the patterns. Figure 3.5 is an example that shows the suffix tree and bad-character shift table constructed by pre-processing patterns "xyz", "rstyz" and "abcdeyz". Figure 3.6 is an example of the search of a sample text "patternrstyz" in a suffix tree.

To begin pattern matching, the shortest of all patterns is left-aligned with the left of sample text (Before Shift) or packet data. The match is then started from right to left character by character. When the test cannot match any character in sample text, the algorithm uses the bad-character shift table for moving the text to the left which may also result in skipping characters. SBMH, Aho-Corasick and Boyer-Moore Horspool

[3]Aho-Corasick pre-process patterns to create pattern search tree in order to support simultaneous search of multiple number of patterns.

[4]Horspool algorithm pre-process patterns in order to gain pattern heuristic information for creating search shift-table.



a	b	c	d	e	r	s	t	x	y	z	*
6	5	4	3	2	4	3	2	2	1	1	4

FIGURE 3.5: Suffix tree and Bad-character shift table for SBMH

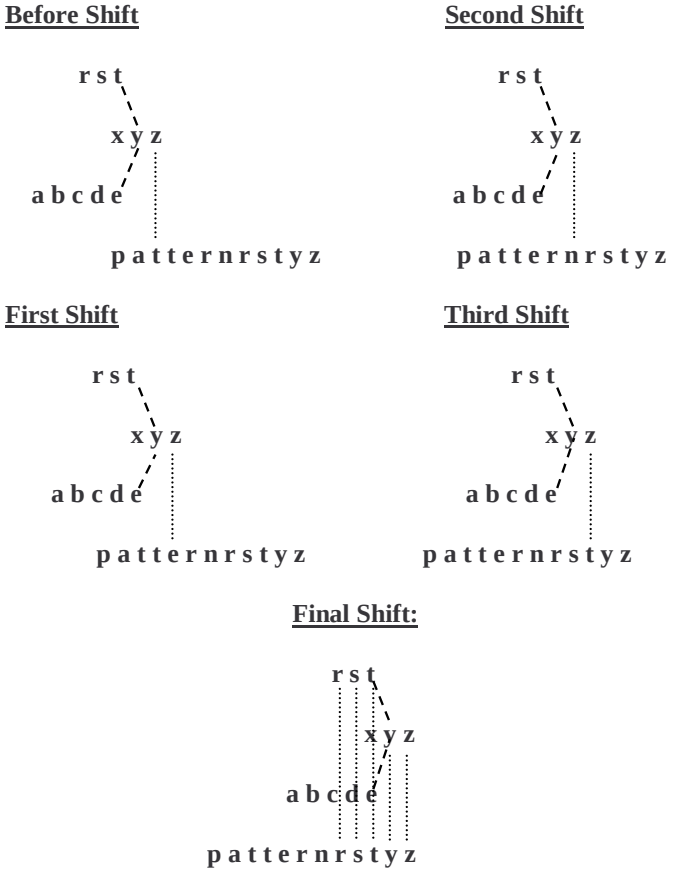


FIGURE 3.6: Example showing pattern search in a text "patternrstyz". Pattern "rstyz" is found in a text in final shift

algorithms performance is tested and compared with different set of Snort's attack rules. Based on the test results they suggested that NIDS like Snort should have different algorithm implementation that trigger depending on the number of rules selected on a packet for evaluation. Boyer-Moore-Horspool if there is only 1 rule, SBMH if there are between 2 to 100 rules and Aho-Corasick if there more than 100 rules. Also for packet traces of web traffic the SBMH algorithm is much better and improves overall Snort performance by a factor of 5. In summary, the novel contribution of combining multi-pattern search with character skipping resulted in improve pattern matching specially for web traffic. However, any pattern matching implementation that uses pattern tree

for multi-pattern search consumes huge amount of memory.

In the same year of 2001, Coit et al also proposed hybrid multi-pattern matching algorithm which they implemented independently in Snort [73]. They combine multi-pattern search of Aho-Corasick and character skip feature of Boyer-Moore in a single algorithm and so named as *AC-BM (Aho-Corasick-Boyer-Moore)*. Like SBMH, AC-BM also pre-process patterns to construct the pattern tree for multi-pattern search support and also gained pattern heuristic information for constructing a search shift-table. However, unlike SBMH which creates a suffix tree, AC-BM pre-process patterns to create prefix tree and also search the packet data for patterns from right to left. Also AC-BM search is supported by has two shift tables instead of one. These shift tables are bad-character shift table and good-prefix shift table. Figure 3.5 is an example that shows the prefix tree constructed by pre-processing patterns "brit", "bribe", "bring" and "brought" and also shows the initial alignment of search text "searchthistext". Figure 3.8 is an example of the searching a text using AC-BM suffix tree applying only good-prefix heuristic.

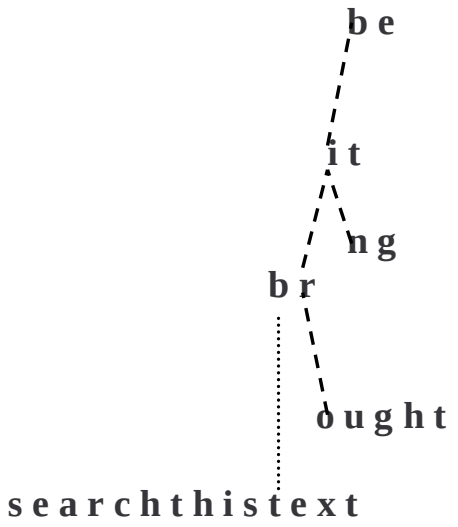


FIGURE 3.7: Prefix tree and text alignment to begin pattern search in AC-BM algorithm

The bad-character heuristic works in the same way as in SBMH. However, the good-prefix shift is a different and complicated process than bad-character heuristic. In figure 3.8 case (a), match test is failed at character 'g'. Text symbol "to" has matched so far with pattern in a tree. The text can be shifted until the next occurrence of "to" in the pattern is aligned to the text symbols "to'. This is a good-prefix shift. Similarly, for case (b) in figure 3.8, the comparison fails at character 'o' of text. There is no other occurrence of "sit" in any pattern. However, since a prefix "si" of the match text "sit" occurrence exists in pattern, the text can be shifted until the next occurrence of "si" in the pattern is aligned to the text symbols "si".

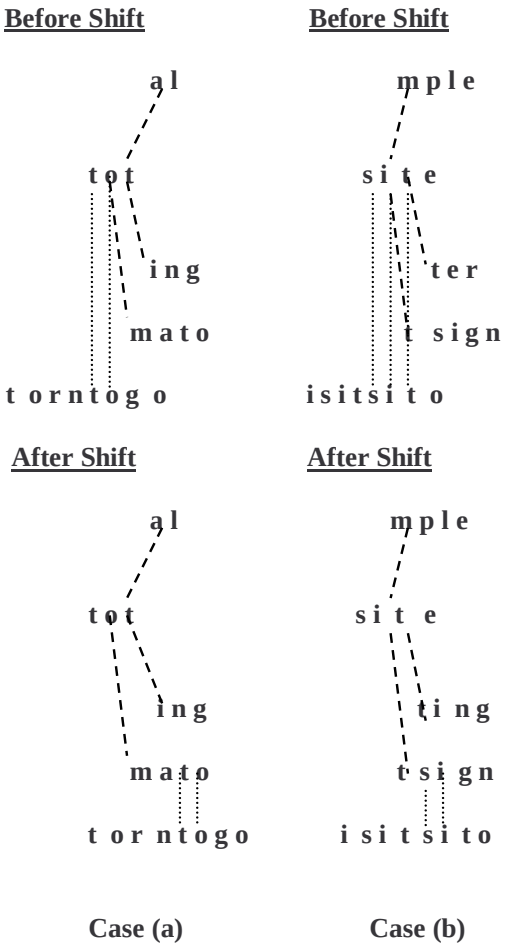


FIGURE 3.8: Search shows good-prefix shift

AC-BM performance is compared with the original implementation of Boyer-Moore. For the combined content (keywords) and non-content attack rules search, Boyer-Moore performed better than AC-BM algorithm. However, when the test were skewed by the elimination of the non-content attack rules, the AC-BM algorithm found superior to Boyer-Moore pattern matching algorithm. AC-BM found 1.31 times faster than Boyer-Moore when tested with 200 Snort content only attack rules and 3.32 times faster than Boyer-Moore when tested with 786 Snort content only attack rules. There is no study identified that provide the test comparison between AC-BM and SBMH.

In summary, the two state of the art SB-NIDS specific algorithms AC-BM and SBMH adopted multi-pattern matching and skip based search technique. The whole idea of hybrid approached proved better search result. No effort has been made to reduce the memory requirement of pattern search tree in these two proposed work. This was addressed separately by Tuck et al. [96] and Marc Norton [97] in a separate study.

3.5.2 Packet Filtering Technique for Pattern Matching in SB-NIDS

The idea of filtering elements (Patterns) in order to reduce the search time first conceived by Burton Bloom in 1970 [5] (See section 5.3.1). He proposed the use of bit-array for quick filtering of the patterns prior triggering any exact pattern matching algorithm for exact matching. If quick filter lookup indicates the pattern presence then it searches with exact pattern matching algorithm, otherwise it simply discarded. This basic idea of filtering is also exploited in other work for optimising pattern matching of SB-NIDS. Some of these work are explained in [74–79]. They can be collectively called as *Pattern Filtering Systems* (Table 3.5).

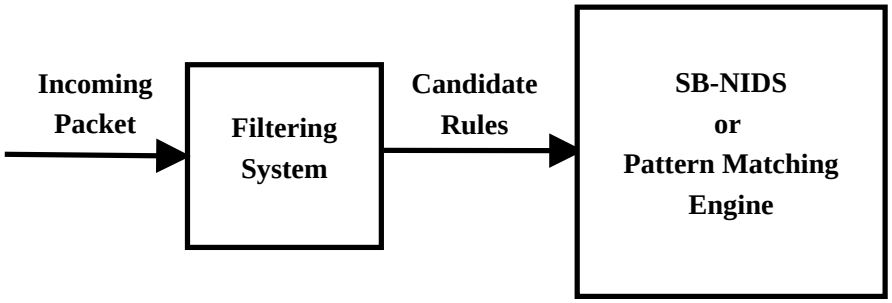


FIGURE 3.9: Block diagram showing typical position of Filtering System for SB-NIDS

TABLE 3.5: Pattern Filtering Systems

Implementation	Authors
Software (General Purpose Processor)	Markatos et al [74] (ExB)
	Anagnostakis et al [75] (E^2xB)
	Antonatos et al [76] (PIRANHA)
Hardware (FPGA)	Attig et al [78] (SIFT)
	Song et al [79] (Snort Offloader)
	Sourdis et al [77] (Prefiltering)
	Gonzalez et al [80] (Shunting)

Initial filtering systems focussed to develop the algorithmic solution of filtering and implemented as a software program (Table 3.5) for general purpose processors [74–76]. Subsequent work optimised filtering algorithms and exploited the processing power of FPGA (Table 3.5) to provide high speed filtering system [77–79]. Table 3.6 shows the details of hardware technologies used for the implementation of these filtering system.

Research Contribution

Markatos et al in 2002 proposed the pattern filtering idea for Snort SB-NIDS [74]. Their filtering system is known as "Exclusion-based signature matching or ExB" that filters or excludes as many patterns as possible so only few remaining patterns are

Table 3.6: Details of Hardware technologies

Authors	Hardware Description
Markatos et al [74])	Intel Pentium IV 1.7GHz, 8-KB of L1 cache, 256 KB of L2 cache and 512MB RAM
Anagnostakis et al [75]	
Antonatos et al [76]	Intel Pentium IV 2.8GHz, 8-KB of L1 cache, 512 KB of L2 cache and 1GB RAM
Attig et al [78]	Xilinx Virtex XCV2000E-8 FPGA
Song et al [79]	
Sourdis et al [77]	Xilinx Virtex II 4000-6 FPGA and Xilinx Virtex IV 40-12 FPGA
Gonzalez et al [80]	Xilinx Virtex-II Pro 30 FPGA

Table 3.7: Advantages and disadvantages of software based pattern matching filtering system

Authors	Advantages	Disadvantages
Markatos et al [74]	Recognised as first ever implementation of filtering technique in SB-NIDS.	Easily become a victim of DoS attack due to slow process of bit mapping in character array for every incoming packet.
Anagnostakis et al [75]	Optimised ExB (E2XB) version now supports the case insensitive pattern filtering.	Still the optimised ExB (E2XB) repeatedly create bit-mapped in character array for every incoming packet.
Antonatos et al [76]	Process patterns only once to create suitable data structure for filtering that can easily implement in hardware.	Provide no support to filter network traffic for patterns less than 4-bytes.
Attig et al [78]	Ability to achieve throughput of 20Gbps using a Xilinx Virtex-IV FPGA.	No support to scan payload content for length 1, 2 or 3-bytes pattern.
Song et al [79]	Packet filtering (Filter firewall), header only rule checking and two level bloom filter on average filters 87% of traffic.	Snort on external PC rather than integrated embedded processor(s) connected with filtering hardware via Ethernet interface.
Sourdis et al [77]	Filtering hardware and Intrusion Detection System (IDS) on a single embedded processing board.	Integrated IDS only performs the rule evaluation on a packet.
Gonzalez et al [80]	An inline filtering system (Shunt) that acts as an Ethernet card to the host that analyse packets using Bro NIDS.	Bro on PC with shunt acting as its gigabit Ethernet card experienced a small fraction of packet drop but drop can become worst on higher data rate interface (10-Gbps).

search using pattern matching algorithm. ExB filtering algorithm is defined as, "For any pattern P with i number of characters c_i, if any i^{th} character c_i of pattern P does not show up in packet payload content T, then the pattern P is not present in packet payload content T." To implement this algorithm, ExB creates the data structure or bit-map of every incoming packet payload and perform the quick check of every pattern character presence in this data structure. The bit-map is created in a character-array

of 256 indexes which always set to zero before bit-mapping using *bzero()* C-language method. Figure 3.10 shows the ExB's algorithm bit-mapping process of a sample text "1000poundsinnetworkpacket" and figure 3.11 shows the filtering of patterns "100dollar", "1001pounds", "1000pounds" and "nomoney" in a ExB created bit-map of a sample text "1000poundsinnetworkpacket" in figure 3.10.

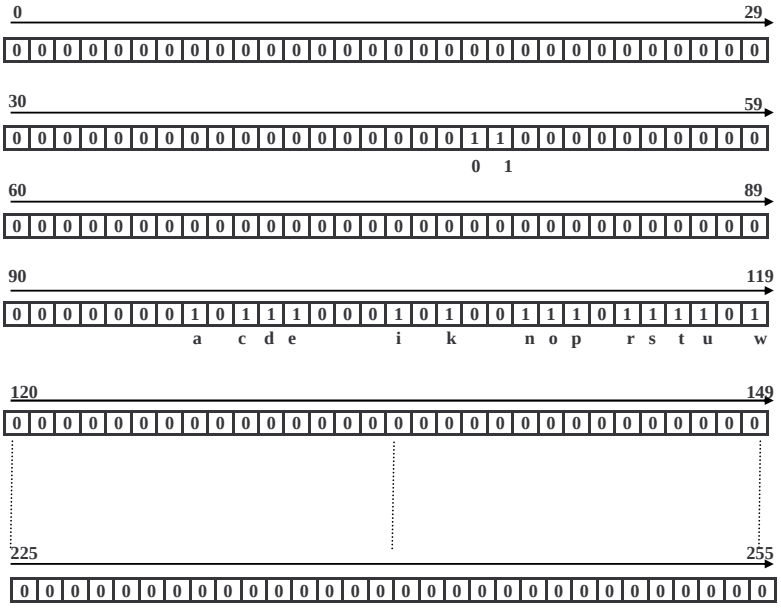


FIGURE 3.10: Pre-processing in ExB of a text "1000poundsinnetworkpacket"

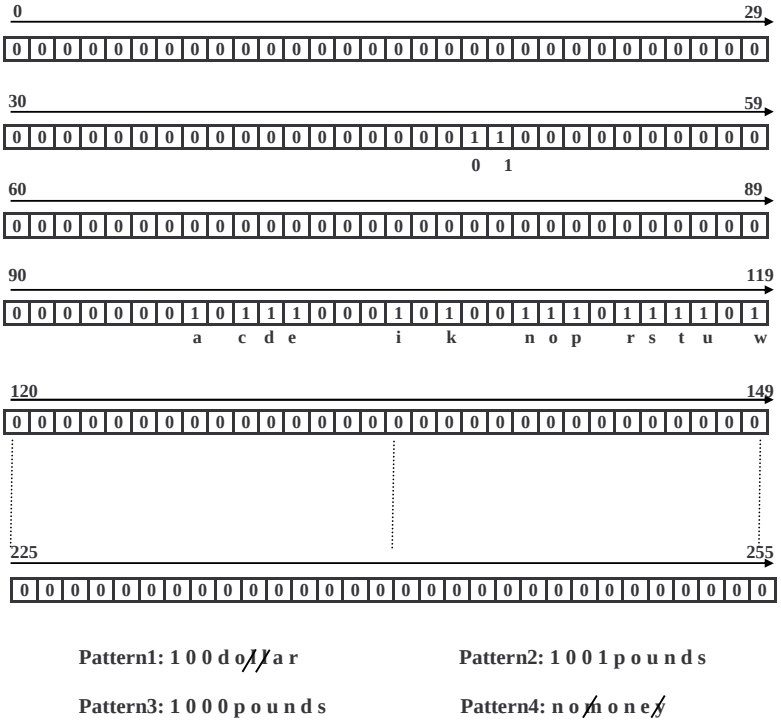


FIGURE 3.11: ExB algorithm searching patterns in a text "1000poundsinnetwork-packet"

In figure 3.10, character array index with a value 1 indicates the presence of particular character in a sample text "1000poundsinnetworkpacket". For example, ASCII value of character 'w' is 119 so the algorithm sets the corresponding array index to 1. Figure 3.11 which shows the pattern filtering , the mark (/) on characters of patterns shows that the corresponding character array index is not set to 1 during bit-mapping which indicates the sample text does not contain that character. Therefore, this pattern will be excluded for pattern matching in Snort using one of the pattern matching algorithms such as Boyer-Moore. For patterns "1001pounds" and "1000pounds" all characters are found in a bit-map so both pattern will be search in Snort with pattern matching algorithm. Pattern "1001pounds" with pattern matching in Snort would result as a non-match pattern which in other words is a false positive and a weakness of this filtering algorithm also acknowledged by authors. Instead they suggested the use of 13 bits character array which they consider a good trade-off between false positive and lower memory usage. The novel contribution which advances the state of the art is the first ever filtering system for SB-NIDS in order to lower the invocation of computationally demanding Boyer-Moore pattern matching algorithm in Snort. The weakness of this work is the high compute time require for bit-mapping, the method which execute for every incoming packet and an easy target of DoS service attack such as smurf attack (Section 2.2.1). This weakness is recognised and also improved by Anagnostakis et al in 2003 by introducing ExB algorithm enhancement known as E^2xB [75].

E^2xB is an optimised version of ExB that provide the faster bit mapping process, support for case-insensitive matching and is tested with wider set of experiments. The faster bit mapping process is improved by removing the overhead associated with initialising a character array or clearing of 256 bytes character array (all array index should set to 0 before processing packet data) using (bzero() C-language method). Now in E^2xB, the array index for corresponding character ASCII value is marked with unique *packet-ID* instead of 1 which serves the purpose of indicating the presence of particular character in a packet data. E^2xB algorithm performance is also compared with state of the art SBMH and Boyer-Moore pattern matching algorithm. It was concluded from the test results that E^2xB consumes less search time than Boyer-Moore and SBMH for all network packet traces except one. Improvement of E^2xB is nearly 25 % and in some cases can be as high as 36.1 % over SBMH and Boyer-Moore. In only one case of experiments E^2xB found worse than Boyer-Moore by 8 %. In summary, ExB and E^2xB are one of the first pattern filtering system for Snort which advances the state of the art. Performance of E^2xB is better than two state of the art pattern matching algorithms, even optimised E^2xB processes every packet to create bit-map instead of one bit-map of pattern for all packets which is more appealing and practical to get the better result.

Antonatos et al in 2005 proposed *PIRANHA*, a filtering system for Snort suitable to implement in hardware [76]. PIRANHA filtering algorithm is optimised and better than ExB. The algorithm searches only 4 bytes rarest substrings of patterns in a packet data and if found in packet then only those patterns are fully search with Snort pattern matching algorithm. Rarest substring of pattern reflects the least number of times that a specific substring exists in all patterns. PIRANHA algorithm is implemented with Hashtable that is more suitable for filtering. It first finds all the 4 bytes substring in all patterns and selects only one rarest 4 byte substring to represent each pattern in hash table. Figure 3.12 shows the examples of PIRANHA arranging the patterns "filteringprocess" and "filterisneverbad" in a hashtable for filtering and figure 3.13 shows the pattern filtering process for a sample text "verygoodfilteringprocess".

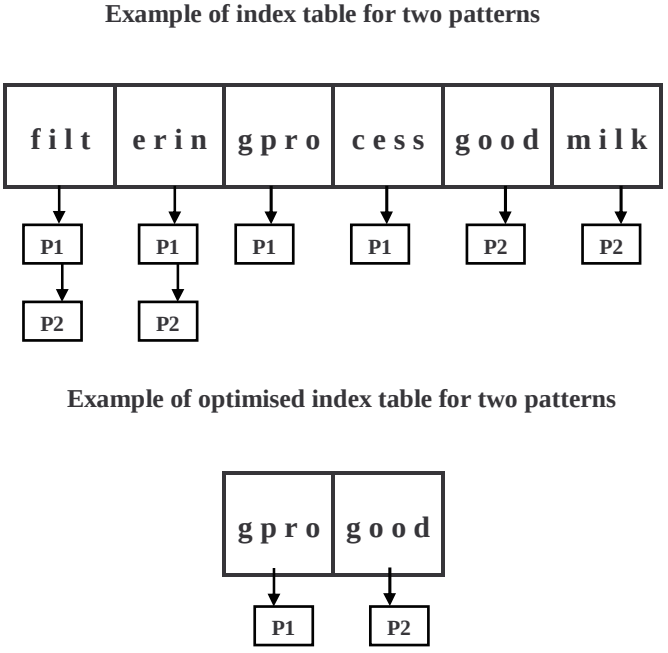
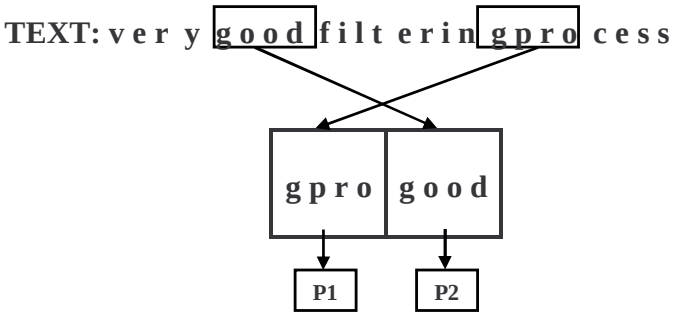


FIGURE 3.12: Example of pre-processing of patterns "filteringprocess" and "filteringoodmilk" in PIRAMHA



PATTERN1 (P1): filteringprocess **PATTERN2 (P2): filteringoodmilk**

FIGURE 3.13: Example of searching text "verygoodfilteringprocess" for patterns "filteringprocess" and "filteringoodmilk" in PIRAMHA

The rarest substring selection and association to patterns in hashtable is a two stage process. At first the un-optimised hashtable is created which would result in slow pattern filtering performance due to high number of memory accesses but low false positive rate. In the second stage, the optimised hashtable is created by selecting only one 4 byte rarest substring to represent each pattern which would result in high speed pattern filtering performance but high false positive rate. Filtering is more straight forward process than hashtable setup. For every incoming packet, each 4 byte sequence of packet payload content substring is checked in the optimised index table to find occurrence of any rarest 4 byte pattern substring. If any 4 byte rarest pattern substring matches with the 4 byte packet data substring then algorithm compares the last 2 bytes of pattern with the corresponding 2 bytes of packet content. In case this also result in match then pattern is send to Snort to compare it with a corresponding bytes of packet content using pattern matching algorithm. PIRANHA performance is tested and then compared with some state of the art algorithms which include Mu-Wanber multi-pattern matching algorithm (MWM) and E^2xB [98]. The test results show consistently better performance than these two algorithms. With eight different network packet traces, PIRANHA performance is between 10 % to 23.50 % better when compared to other two algorithms. PIRANHA also has low memory requirement. For full Snort attack rules (2500 number of Snort rules in 2005), PIRANHA only consumed 37 MB of memory while MWM consumed 45 MB, Aho-Corasick (AC) consumed 140 MB, variants of AC like Marc Norton [97] AC-BANDED consumed 96 MB, Tuck et al. [96] AC-Bitmap and AC-Path needs 20 MB and 15 MB of memory respectively. In summary, PIRANHA pattern filtering algorithm is more optimised than ExB algorithm because it only processes patterns once for pattern filtering. Also it is easier to implement in hardware due to simple Hashtable implementation. It also has better performance than other state of the art algorithm but has a drawback of not supporting the patterns of less than 4 byte length which are approximately 400 in numbers in June 2009 release of Snort attack rules.

Song et al in 2005 proposed the *Snort Offloader* pattern filtering system using a combination of hardware and software processing platform for improving packet analysis speed of Snort [79]. The hardware side of hybrid platform is the reconfigurable hardware (Xilinx Virtex XCV2000E-8 FPGA) on which high speed pattern filtering system is implemented in order to reduce the number of pattern search using Snort pattern matching algorithm which is executed on the general purpose processor. Figure 3.14 is the block diagram showing only two main FPGA based hardware filtering system modules.

The two hardware modules are: Active packet filter (APF) and Passive packet filter (PPF) is implemented on Xilinx XCV200E FPGA. APF is loaded with traffic flow

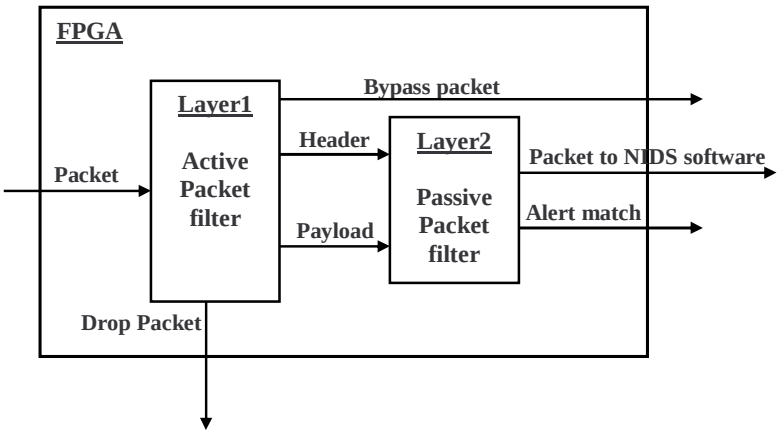


FIGURE 3.14: Block diagram of Snort offloader showing two main hardware modules

information for active bypassing or blocking of packet (Packet filter firewall 2.3.2) and PPF is loaded with 2600 attack rules comprises of patterns and other packet header values for the purpose of filtering. APF is the first layer or module that receives the packets from the network interface and checks key header values (mainly IP addresses and Port numbers) for bypassing or blocking for certain traffic flow. At the same time, PPF inspect the packet header and payload against attack Snort rules with the help of its two sub-modules: Header classification and Two-level Bloom filter. First packet payload content is searched for pattern presence by computing eight hash values per substring in just 2 clock cycles to check the corresponding index value of the first-level 16-Kbit Bloom filter which takes another 3 clock cycles. If Bloom filter report possible presence of any pattern then this pattern combines with attack rule IDs from the header classification module is hashed together for checking in second-level Bloom filter index. If the second-level Bloom filter also reports match, then packet is forwarded towards the software executing Snort matching pattern algorithm. PFF also match the 144 number of header only attack rules (No patterns) in packet header classification hardware sub-module, thus further reducing processing load on Snort. This pattern filtering system is tested with traces taken from Washington University network shows on average 87 % of network traffic is reduced or filter. Authors also claim that the filtering system can successfully operate to scale its operation up to 10 Gbps but no such claim proof is provided with the help of any experiment results. In summary, this is the first ever FPGA hardware based filtering system which performs pattern filtering as well as the header only attack rules processing in FPGA. Its processing speed is much higher due to hardware based implementation and much better than previously discussed software based pattern filtering system.

In 2005, Attig and Lockwood proposed *SIFT: Snort Intrusion filter for TCP* which is very similar to Snort Offloader [78]. SIFT like Snort Offloader is developed with a combination of hardware and software processing facility. The filtering system is

implemented on FPGA hardware with Xilinx Virtex XCV2000E-8 FPGA in order to reduce the amount of network traffic forwarded to Snort SB-NIDS for pattern search in packet payload. Snort is executed on general purpose processor (AMD Athlon MP 2600+ with 3 GB RAM). SIFT also performs the header only attack rules on a packet in FPGA hardware. The difference in the approach is that SIFT uses five Bloom filter engines (16 Kbits vectors in FPGA block memory) for checking substring of packet payload content for the presence of attack patterns. These is carried out by calculating eight hash values on every substring of packet data and perform Bloom filter index lookup for corresponding hash values. If the corresponding Bloom filter indexes are all set to 1 that indicates the pattern presence with certain false positive probability, then the packet is forwarded to the Snort for comparison using Snort pattern matching algorithm such as Boyer-Moore or Aho-Corasick. The filtering system is implemented with 2464 Snort attack rules and able to operate at 80 MHz, provides a throughput of 2.5 Gbps. It processes 4 bytes per clock cycle and also filters between 86 % to 96 % of network traffic for common network protocols (TCP, UDP, ICMP, and IP). In summary, SIFT and Snort Offloader provide very similar packet filter results. It's only weakness is lack of support to scan packet payload content for 1, 2 and 3-bytes patterns which constitute around 400 patterns of June 2008 release of Snort attack rules. In comparison to very similar Bloom filter based pattern matching hardware design on FPGA presented in this thesis (Section 5.3), SIFT operates with higher operating frequency and so provides higher throughput. The lower pattern matching hardware operating frequency of the pattern matching hardware design in this thesis limitation imposed by the hybrid hardware-software processing platform which does not enable to synthesised the FPGA design with MicroBlaze soft core processor of more than 50MHz frequency. Another reason of attaining such a higher throughput by SIFT is that it offered only pattern filtering system which does not perform any false positive patterns pruning. Furthermore, the number of attack rules is nearly 3 times lower in this pattern filtering system which directly affects the throughput.

In 2006, Sourdis et al proposed a FPGA based packet filtering system with integrated Intrusion detection system (IDS) on the same FPGA [77]. This IDS consists of payload matching and specialised processing engine. This specialised processing engine performs regular expression and static pattern matching for Snort attack rules evaluation. The filtering purpose in this system is to reduce the evaluation of number of rules search per packet in IDS engine. Figure 3.15 shows the filtering process.

All packets first pass through the filtering hardware *Field Extractor* module where packet header fields and payload content are extracted. Next the packet header is feed into the filtering hardware *Header matching* module which perform key packet header field (Protocol, IP address and Port) match with rule header and reported the successfully match

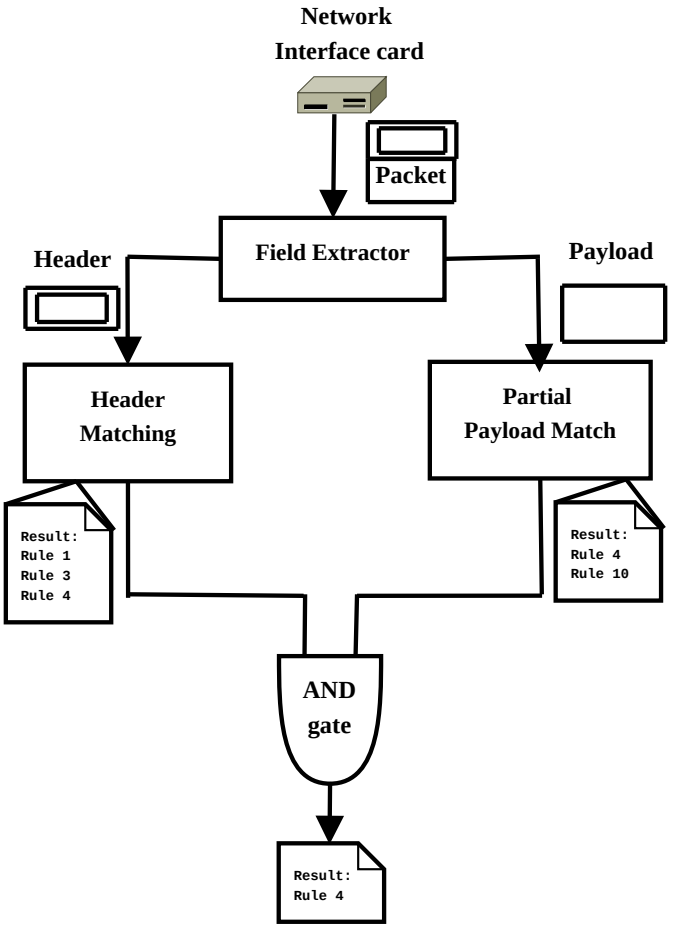


FIGURE 3.15: Packet processing flow in filtering hardware

rules. Simultaneously, the packet payload content also feed into the filtering hardware *Partial payload match* module which match the payload content constant number of prefix bytes (Between 2 to 10 bytes) with patterns from attack rules and report the successfully match rules. Output from both modules are then AND (The AND gate) and then final list of rules are then reported to integrated IDS for full matching of attack rules. The filtering system is implemented with 3191 Snort rules (2271 number of patterns) using two different FPGA families: Xilinx Virtex2-4000-6 and Virtex4-40-12. With Virtex2-4000-6, packet filtering hardware is able to synthesised up to a clock frequency of 335MHz (8-bits/clock cycle) giving an effective throughput of 2.7 Gbps and with a clock frequency of up to 303 MHz clock (32 bits/clock cycle) giving 9.7 Gbps of throughout. With Virtex2-4000-6, the filtering hardware is able to synthesised with a clock frequency of up to 335 MHz (8 bits/clock cycle) giving an effective throughput of 4.0 Gbps and with clock frequency of up to 303 MHz (32 bits/clock cycle) providing 14.0 Gbps throughout. The IDS part which has a coprocessor for pattern matching is able to support 2.0 Gbps which is also very high throughout. However, an IDS in this system is just a rule evaluation system and an additional layer of packet analysis software system is needed to perform protocol analysis and stateful packet inspection that

can only complement the lack of basic function of this IDS. In summary, it is the first ever integrated FPGA filtering system with IDS which lacks in basic IDS functionality but able to able to reduce the significant number of rules evaluation in IDS from 45 rules on average per packet to 1.8 rules on average per packet.

TABLE 3.8: Hardware based pattern filtering

Authors	Number of Rules	FPGA	Throughput
Attig et al [78]	2464	Xilinx Virtex XCV2000E-8	2.5 Gbps (4 bytes/cycle at 80MHz
Song et al [79]	3600	Xilinx Virtex XCV2000E-8	10.0 Gbps (Estimated throughput)
Sourdis et al [77]	3191	Xilinx Virtex-II 4000-6	9.7 Gbps (4 bytes/cycle at 303 MHz
		Xilinx Virtex-IV 40-12	14.0 Gbps (4 bytes/cycle at 303MHz

In 2007, Gonzalez et al proposed the first ever inline packet filtering system using Net2FPGA 2.0 development platform. Net2FPGA platform has four gigabit Ethernet interface connected to a Xilinx Virtex-II Pro 30 (XC2VP30) FPGA [80]. The filtering system acts as an Ethernet card to the host machine that has a Bro IDS installed to perform packet analysis. When a network packet arrives the filtering system chooses from one of three possibilities: a) *Forward* the packet to the opposite interface (Pass packet without inspection) b) *Drop* it (Packet is identified as an attack) or c) *Divert (Shunt)* the packet towards the host (Performing packet analysis). The filtering system carried out these operations with the help of Bloom filter which is programmed with malicious packet key header values (IP addresses, port numbers and Protocols). First the filtering system hashed every incoming packet header key values (IP addresses, port numbers and Protocols) and perform Bloom filter index checking with corresponding hash values. If Bloom filter lookup result in a match found then Hashtable is checked. The hashtable entry may include an action (forward, drop or shunt) to take on packet with a priority defined from 0 to 7. A priority encoder then selects the highest priority action and performs the corresponding action on the packet. The test result of filtering hardware with network traces taken from University of Berkeley network shows that the in best case, 88 % packets are forwarded to pass through network without analysis and in worst case, the percentage of network packets passed without analysis dropped to 43.8 %. In summary, this is the first ever inline packet filtering system that does not simply forward the packet to IDS but take appropriate action and so advances the state of the art.

In summary, the idea of filtering system seems convincing for reducing computationally load on pattern matching in SB-NIDS. It is more effective when implemented in FPGA as

proposed by Song et al. [79] and Attig and Lockwood [78]. This is made further effective and efficient by Sourdis et al. [77] and Gonzalez et al. [80] by presenting a FPGA based filtering system with integrated NIDS on the same tightly coupled hardware architecture.

3.5.3 Pattern matching using High Performance Computing Platform

To optimise the pattern matching speed for Snort, the high performance computing platform has been the subject of great interest for almost a decade. The two high performance platform widely explored for this purpose are *reconfigurable hardware (FPGA)* and *Network Processor*. ASIC has also been used for pattern matching optimisation, such as Kumar et al ASIC based regular expression based pattern matching which is implemented using finite state pattern machine approach [99]. Overall most of the effort is implemented with FPGA or Network Processor. These can classified further by the chosen approaches of implementation, such as Hashing, Bloom filter and State machine (Finite Automation or Non-Finite Automation)) that efficiently utilised the high performance platforms processing power and resources. Table 3.9 summarise some of these work implemented using FPGA [84, 86, 88, 89, 100–104] and network processors [93, 105][5].

All pattern matching design in table 3.9 have one thing in common that they implemented and tested using Snort attack patterns. There are other pattern matching approach implemented using FPGA and more closely related to the work in this thesis (Section 5.3). These work along with others closely related work to this thesis will only be discussed in detail. These work are Bloom filter based pattern matching using FPGA [81, 82, 106] and Snort rule processing optimisation which can be collectively called as Snort Rule Processing System [85, 87, 90–92, 94]. Snort Rule Processing System involved packet processing that include packet classification (Rule selection) and packet analysis (Packet header and payload check) which are implemented using FPGAs [85, 90, 94], hybrid hardware-software embedded processing platforms [87] and Network Processors [91, 92]. (Table 3.10) shows the hardware used in these implementations.

Research contribution

In 2002, Gokhale et al proposed rule processing system for Snort using hybrid hardware-software embedded processing platform [87]. They have written the *Rule compiler* software module and the *Rule processor* hardware module. Rule compiler reads the subset

[5]This table illustrate few mostly cited work.

TABLE 3.9: Some pattern matching implementation on FPGA and Network Processor

Hardware Architecture	Authors	Summary
FPGA	Sourdis and Pnevmatikatos [86]	Pattern matching implementation using Content Addressable Memory (CAM) for searching patterns in network packets.
	Baker et al [100]	A pattern matching hardware architecture using Brute-force search approach of pattern matching for SB-NIDS.
	Tan and Sherwood [88]	A hardware design that search patterns in packets in parallel with multiple state machines created of malicious patterns.
	Jung et al [89]	An implementation that converts pattern state machine into state transition table for parallel pattern search in packets.
	Brodie et al [102]	A pattern matching of regular expression that matches multiple patterns concurrently using state machines.
	Sourdis et al [84]	A pattern matching using hashing tree and string comparator hardware circuit.
	Mitra et al [103]	PCRE (Perl Compatible Regular Expression) Engine implementation using state machine based pattern search.
	Kennedy et al [104]	Modified Aho-Corasick (State machine) implementation resulting 98% reduction in memory consumption small enough to fit in the on-chip FPGA memory.
Network Processor	Bos and Huang [93]	Aho-Corasick (State machine) implementation for searching patterns in packet in parallel using multiple processors.
	Piyachon and Luo [105]	An implementation that modify Aho-Corasick (State machine) into multiple bit-level state machine for searching pattern in packets using multiple processors in parallel.

of Snort attack rules and creates a representation of rule fields suitable for hardware implementation. Using these rule fields, the Rule compiler then initialises the Content Addressable Memory (CAM) content for packet header and payload checking. This checking is carried out by the Rule processor which compares packet header fields and contents with the contents of rule using parallel comparison logic of CAM. Once the Rule processor complete rule checking and output the comparison results then the Rule compiler process carries out the rule match result by raising the alarm for the attention of network. This system is implemented on two Xilinx Virtex-1000 (XCV1000) FPGA on a SLAAC1V board. The hardware designed is able to synthesised up to 66 MHz, giving an effective throughput of 2.0 Gbps when tested with DARPA Lincoln lab network traces for intrusion detection and testing with unspecified number of rules [107]. However, it can estimate from the year of paper publication that the number of Snort rules used for experiment might be around 1500 to 2000 which are very low in number.

Table 3.10: Hardware Details of development platform

Authors	Hardware Description
Gokhale et al [87]	2 * Xilinx Virtex 1000 XCV1000 FPGA on the SLAACIV board.
Liu et al [91]	Vitesse IQ2000 (VSC2100) Network Processor with four 200 MHz RISC CPU packet processing engine.
Attig and Loockwood [94]	Xilinx Virtex 2000E XCV2000E FPGA on the Field Programmable Port extender platform.
Yusuf et al [85]	Xilinx Virtex-II Pro XC2VP30 FPGA on Xilinx University Program board.
Caruso et al [92]	Prototyped on Digilent Spartan-3 board with 3-picoCPU processor.
Cho et al [90]	Xilinx Spartan-3 XC3S400 FPGA.

FPGA device utilisation summary for this design also shows that the complete rule processing system occupies total 34 % of the FPGA logic slices of Xilinx Virtex XCV1000 FPGAs (8396 out 24576) which is acceptable because it comprises of rule header and payload checking and on two FPGAs. In summary, this rule processing system is the first ever proposal to addressed optimisation of Snort rule evaluation. A complete rule evaluation system without integrated SB-NIDS is carefully designed to achieve a decent throughput of 2.0 Gbps.

Liu et al in 2004 proposed a skip based prefix matching algorithm on Network Processor [91]. This skip based algorithm which simplifies Snort rule processing is based on simple reasoning: For any arbitrary pattern P, if packet stream with T number of sequential bytes do not contain prefix of arbitrary pattern P, then the number of bytes equal to the length of P can be skipped during searching. If in case prefix of P found, then it is highly likely that the pattern is present in a packet stream. To confirm the pattern presence a hashtable named as Rule Hashing Table (RHT) is implemented that contains attack patterns and associated Rule IDs. The hash value is computed on the suspected substring that previously matched with prefix pattern and RHT is lookup in order to identify Rule. This algorithm implemented on Vitesse IQ2000 network processing board that has four 200 MHz RISC processors with each processor has 2 KB of internal memory and a shared 512 MB Direct Rambus Dynamic Random Access Memory (RDRAM). The algorithm performance is compared with the state of the pattern matching algorithm which shows that this algorithm is more efficient than set-wise Boyer-Moore Horspool [72], Aho-Corasick [55] and Wu-Manber [98] when length of smallest pattern (LSP) search in the experiment is less than 4 (LSP $\leq$ 4) (35 % of Snort pattern length were 4 or less when they tested the system.) out of the 1942 Snort attack rules. In summary, it is a rule processing system that presents novel skip based pattern matching algorithm. The experimental results show the algorithm performance better than other state of the art algorithm only when pattern length 4 or less is search

in packets. This is not a promising result with respect to the current Snort attack rules number which is around 9000 which comprises of approximately 6800 patterns with over length 4 patterns.

Attig and Lockwood in 2005 developed a framework for implementing a Rule processing system in FPGA on University of Washington field programmable port extender (FPX) platform [94]. The rule processing framework main components are packet header processing and packet content scanning modules which is implemented on a Xilinx Virtex 2000E (XCV2000E) FPGA. Their implementation details are not well defined because the stress is put in presenting the framework architecture. This system is implemented with 2464 Snort rules which is able synthesised up to 80.6 MHz operating frequency, giving a rule processing throughput of 2.56 Gbps. The device utilisation summary shows the usage of only 25 % FPGA slices (4,832 out of 19,328) which is an efficient design. In summary, a rule framework completely implemented on FPGA is presented which claim to support for up to 32,768 Snort rules but does not provide any proof in the form of test results. Also, an experiment for worst case traffic the framework throughput falls to below 500 Mbps which is a clearly a bottleneck on gigabit data rate network.

Yusuf et al in 2006 proposed *UNITE* which is a rule processing system or engine [85]. UNITE performs packet header classification (Rule selection) and packet payload analysis utilising CAM which provide parallel comparison logic for matching packet header and payload for rule selection and pattern matching. This design is implemented on the Xilinx University Program (XUP) board which has a Xilinx Virtex-II Pro XC2VP30 FPGA. UNITE is implemented with 74 Snort attack rules and achieved an operating frequency of 350 MHz with a packet processing throughput of 2.84 Gbps. In summary, UNITE is a rule processing system but no packet header checking part of rule processing which is a crucial weakness with respect to network security. It provides higher throughput but only when implemented with 75 attack rules which has now soared to around 9000 attack rules and would significantly reduced the overall throughput of UNITE.

Caruso et al in 2007 proposed SPP-NIDS which performs parallel search of Snort attack rule using clusters of processors [92]. This system is prototyped on Digilent Spartan-3 embedded processing board which has 3-picoCPUs (Processors). In SPP-NIDS, attack rules are distributed and stored in picoCPU internal RAM. On receipt of packet in picoCPU, each processor analyses the packet for the subset of rules stored in its local memory. The prototype system performance is tested with 2124 Snort attack rules. It shows that SPP-NIDS is able to analyse up to 100 Mbps network data rate which is very low. The theoretical performance estimation of SPP-NIDS with 30-picoCPU operating in parallel at 200 MHz frequency and with 1600 Snort rules shows maximum achievable throughput of 53 Mbps. In summary SPP-NIDS is the lower performance

rule processing system which is not feasible to be deployed on gigabit data rate network. Its performance may further degrade if implemented with 9000 number of Snort rules which would significantly increases the average number of rules checking per packet.

Cho et al in 2008 observed that most of the pattern of Snort attack rules can be searched in a packet content in parallel and ideal to implement using FPGA parallel logic resource [90]. They developed a parallel search engine which comprises of a hardware module called rule units. Each rule units implement the logic for a single Snort rule signature which comprises of rule header matching unit and comparator unit to compare pattern with the packet payload content. Their pattern matching system was unable to synthesise successfully only 2000 Snort attack rule. This is due to the inefficient design which creates the static pattern comparator which requires atleast 66,000 LUTs for 8 bit and over 260,000 LUTs for a 32 bit comparator. They improve this design by leveraging on the hardware architecture and the data-specific optimisations. The optimisation decreased 50 % of the logic area which is achieved by changing a design to the memory based pattern tree search approach. Their final design implementation on Xilinx Spartan-3 XC3S400 FPGA with 2000 Snort attack rules achieved a sustained throughput of 1.6 Gbps. In summary, this rule processing system is an efficient design but provide no header check facility which is considered an incomplete implementation. It provides decent throughput 1.6 Gbps suitable for gigabit Ethernet data rate traffic.

Table 3.11: Snort Rule Evaluation Systems summary

Authors	Rule Selection	Header Check	Payload Check	Snort Rules (Number)	Throughput
Gokhale et al [87]	*X*	✓	✓	1500-2000	2.0 Gbps
Liu et al [91]	*X*	*X*	✓	1942	Unspecified
Attig and Loockwood [94]	*X*	✓	✓	2464	2.5 Gbps
Yusuf et al [85]	✓	*X*	✓	74	2.8 Gbps
Caruso et al [92]	*X*	✓	✓	2124	100 Mbps
Cho et al [90]	*X*	*X*	✓	2000	1.6 Gbps

In summary, state of the art rule processing systems are discussed. All rule processing system are discussed are not part of SB-NIDS and so are incomplete systems. The issue has been addressed in this thesis which proposed an integrated rule processing system with NIDS on the embedded processing platform which advances the state of the art (Section 5.3). The proposed designed also gives throughput of 1.85 Gbps for pattern matching in FPGA which is lower in comparison to the design presented by Yusuf et al. [85], Attig and Lockwood [94] and Gokhale et al. [87]. The main reason of lower throughput is due to the restriction imposed by the embedded processing platform used in development which does not allow synthesising FPGA hardware design along with MicroBlaze soft core processor for more than 50 MHz operating frequency. This

limitation can easily be overcome by synthesising a design to higher grade FPGA like Xilinx Virtex 7 which would easily support much higher operating frequency. The rule processing implementation presented in this thesis occupies 26357 FPGA slices (26357 out of 33792 slices) which is higher in comparison to the hardware design discussed above but performs 16 parallel rule searches and supports around 9120 attack patterns which supersede every implementation.

There are some other systems which are not a rule processing systems but are related to this thesis work in a way that those work also used the Bloom filter [5] data structure for pattern matching implementation. These FPFA hardware design are now discussed.

Dharmapurikar et al. [81] in 2003 proposed a Bloom filter engine for deep packet inspection (Pattern matching) on FPGA. Each engine has a separate Bloom filter programmed with patterns of Snort attack rules for quick pattern checking in packet payload. When packet payload content is streamed through the system, the hash values are computed on each substring of packet content and corresponding Bloom filter index are checked. If Bloom filter lookup result in a pattern presence then it is further checked with analyser hardware module. The analyser module is comprises of Hashtable where patterns are stored for comparison with a substring using hardware comparator circuit. This step is necessary as Bloom filter gives matching result with some probability of false positive. The functional prototype with a single Bloom filter engine is implemented on Xilinx XCV2000E FPGA on the field programmable port extender (FPX) platform with only up to 32 bytes pattern is able to achieve an operating frequency of 81 MHz. The FPGA logic resources summary also shows that this design is efficient which only occupies 14 % of available FPGA logic resources and 35 block RAM. The test results show that this system performs packet analysis of up to 2.46Gbps throughput. The test results also revealed the implementation weakness which inefficiently searches any pattern over 16 bytes length. To overcome this issue and add support for longer patterns, Dharmapurikar and Lockwood in 2006 extend this design with Aho-Corsaick based multi-pattern matching algorthm [106]. In summary, this is considered the first ever implementation of pattern matching for SB-NIDS using Bloom filter which advanced the state of the art. The hardware architecture is also an efficient implementation apart of the design inefficiency to deal effectively with longer patterns.

Nourani and Katta in 2007 proposed a Bloom filter based hardware architecture for pattern matching on ASIC [82]. Streaming data is passed through the Bloom filter accelerator hardware module which computes the hash value on substring and query the Bloom filter index which may result in false positive match. In case of match the dispatcher hardware module pass the substring to parallel hash engine that perform the false positive check by comparing the matched pattern with the pattern stored in

hashtable. The design was synthesized using Synopsys design compiler using library files from Artisan targeting the 180 nm Taiwan Semiconductor (TSMC) fabrication process and can operate at speed of 250 MHz given an effective throughput of 1000 Gbps that is very higher than all other implementations. In summary, it is a high speed ASIC pattern matching system which supports extremely high throughput.

In summary, the Bloom filter based pattern matching approach is ideal for excluding major part of network traffic for full analysis using computationally demanding pattern matching algorithms. Its straight forward implementation also consumes less memory and enable high speed pattern searching on FPGAs. The only implementation issue is the high number of hash value computation which affects the overall throughput of the system. This issue has addressed in this thesis (Section 5.3).

3.6 Chapter Summary

This chapter began with an introduction of state of the art SB-NIDS implementations which are explained in great detail in section 3.3. The state of the art is divided into categories for clear understanding of state of the art implementations and contributions made to advance the state of the art.

Chapter 4

Proposed System Architecture

Intrusion Detection System (IDS) is currently considered the most reliable and standard solution of the internet security. One of its best examples is the EINSTEIN-2 [108] project of the United States of America (USA) government to protect the government infrastructure of computer networks hosting crucial data. EINSTEIN-2 is a Signature-based Network Intrusion Detection system (SB-NIDS) that uses the set of attack signatures or patterns to detect malicious activity on a network. Developing such a sophisticated solution is a complex task and requires in depth understanding of network security issues, network defence technology and processing technology requirement. This chapter looks into some of these issues in order to lay the foundation of SB-NIDS prototyping and optimisation (Chapter 5).

4.1 Chapter Roadmap

The rest of the chapter is divided into two major sections which are outlined as follows:

- In section 4.2, an overview of SB-NIDS is presented with a focus on its packet analysis architecture and features in order to estimate requirements of developing an improved SB-NIDS prototype.
- In final section 4.3 Snort SB-NIDS software architecture is discussed in detail which helped in outlining prototyping challenges. These challenges also helped to determine the appropriate processing technology for prototyping.

4.2 System Description

This section includes description of SB-NIDS architecture and SB-NIDS features necessary to understand the requirements of SB-NIDS prototyping.

4.2.1 Overview

The SB-NIDS prototype should be capable of performing real-time network traffic analysis on Internet Protocol (IP) network. This analysis should at least involve looking for attack signatures in network packets, Stateful Packet Inspection (SPI) and protocol analysis. If SB-NIDS detects any attack in packet it should also raise alarm and inform the administrator. It should also easily deploy on a network point or network backbone.

4.2.2 Architecture

SB-NIDS can be visualised as collection of software components or modules where each component performs different tasks as shown in figure 4.1.

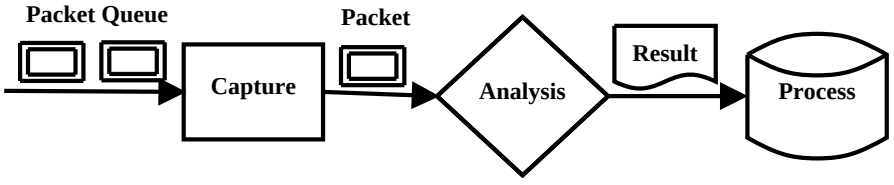


FIGURE 4.1: NIDS Modules

SB-NIDS first captures the data or network packets from network interface in promiscuous packet capture mode. The raw packet data is decode and store into SB-NIDS application memory for carrying out analysis. The analysis is a complex process perform in stages on the dissected data. This include search of attack signatures in packet payload data, SPI to keep track of legitimate network traffic and protocol analysis to detect any protocol based attacks which usually carries out by exploiting vulnerabilities in internet protocols (Section 2.2.1).

In SB-NIDS software the packet analysis is usually perform by multiple analyser components which are configurable in order to enable it to customise and configure for different types of network environment. This configuration involves disable/enable particular type of analysis component, changes in size of memory requires for analysis and updates attack signature database with newly release copy of attack signatures. To apply this configurations SB-NIDS should provide at least simple user interface.

4.2.3 Deployment

SB-NIDS is deployed at a point in network where all network traffic is visible and can be capture for analysis. The general deployment of SB-NIDS is shown in figure 4.2.

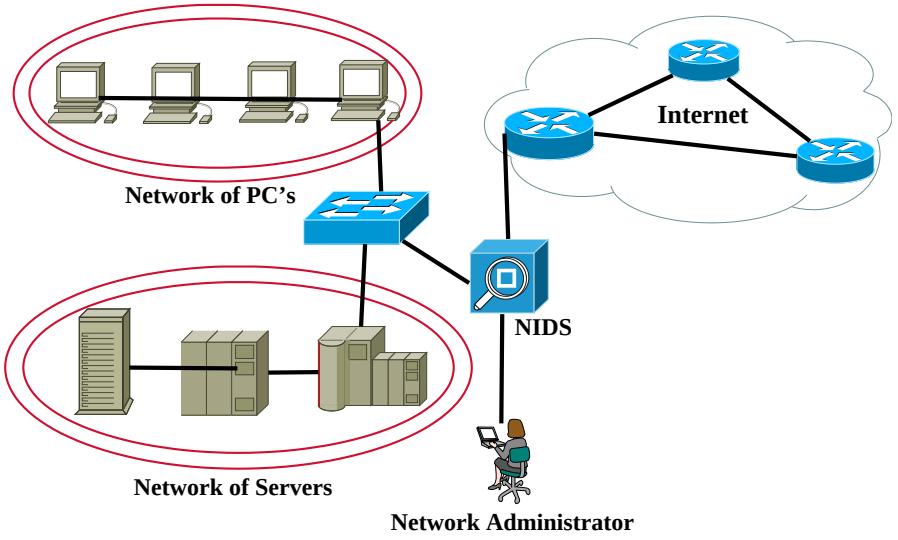


FIGURE 4.2: NIDS Deployment

The machine on which SB-NIDS is installed should have at least two network interface card so it can be connected to a suitable network point or network backbone for network traffic capture and another interface for configurations and management. SB-NIDS is usually connected through a network switch which supports port mirroring which is actually copying of network packets from all ports of switch to the analysis port on which SB-NIDS connects.

4.2.4 Features

SB-NIDS should provide several network security services. Some of the major services are now discussed:

Deep Packet Inspection

Deep Packet Inspection (DPI) is a method that enables inspection of every single byte of every network packet that passes through the network. This means every single byte of packet header as well as packet content from layer-2 through layer-7 (Open System Interconnection (OSI) model) is analyse for attack signature presence using DPI in real time. DPI ensures to detect common attack signatures of malware and therefore essential SB-NIDS features.

Network Protocol Analysis

Network protocol analysis is a process to understand the data and information inside the network packet encapsulated by the network protocols. A typical protocol analysis involved capturing a network packet in real time, decoding of network protocols and analyses of the decoded network packets data. Protocol analysis is perform to detect common network attacks which are carries out by the misuse of network protocols (Section 2.2.1). SB-NIDS should perform network protocol analysis for common protocols whose manipulation can crash network applications and devices. Some protocol based attacks could crashes the SB-NIDS itself such as TCP SYN packet flooding attack on SB-NIDS force it to trigger hundreds of alert every seconds which would result in exhaust of SB-NIDS processing resource (Section 2.2.1).

Stateful Packet Inspection

SPI is crucial for internet traffic monitoring where 90% of traffic is stateful or based on TCP protocol. Lack of SPI feature means SB-NIDS is unable to detect stateful attacks which may present in majority of network traffic. It is therefore essential that SPI should be a part of SB-NIDS. For example SPI enables the SB-NIDS to detect state based attacks such as TCP RST attack (Section 2.2.1).

Configuration Interface

SB-NIDS is sophisticated software that has multiple software analysis components to carry out packet analysis in order to detect many types of complex network attacks. Such SB-NIDS software needs configurable options to customise analysis components and to enable them to operate on different types of network environment. It is essential that NIDS should have some kind of interface to apply configurations and customise the analysis components according to network environment.

Alert Reporting

NIDS or SB-NIDS should accurately and promptly notifies the network administrator if any intrusions or attacks are discover during real time network packet analysis. The reporting should include accurate information about attack such as attacker IP addresses, ports, protocol and timing of attacks. It is therefore essential that NIDS should provide a feature of real time reporting of network intrusion and attacks with basic attack information.

Prevention Capability

NIDS or SB-NIDS softwares only able to detect network intrusions and alert the network administrator to take the appropraite actions. An automatic prevention facility is desirable to prevent attacks without the need of human or network administrator intervention.

4.3 System Prototyping

SB-NIDS prototyping with all desired features is an extremely complex software engineering task which would easily require more than couple of years to create just a basic SB-NIDS prototype model. An alternative technique is a software porting[1]. Porting is comparatively easier and less complex than software development and saved vital prototyping time. However, an in depth understanding of software package internal architecture and other necessary features is required for successfully porting any software.

There were not many choices of good and freely available open-source SB-NIDS packages (Section 2.4.5) that could select for prototyping. The two popular with most of the essential and desirable features are Snort and Bro. Snort is prefer over Bro on the basis of following two reasons:

- Bro is purely experimental NIDS for fine-tuning or improving the packet analysis features whereas, Snort is a widely used open source free available SB-NIDS software package for real time packet analysis.
- Snort in comparison to Bro is the chosen SB-NIDS for research and development in academia mainly for optimising packet analysis speed performance.

4.3.1 Snort

Snort is a SB-NIDS software package capable of real-time detection of network intrusions and attacks using attack signature, SPI and protocol analysis.

History: Snort first became available on December 22, 1998 on Packet Storm website (http://www.packetstormsecurity.com) as network Packet Sniffer application. This consists of 1,600 lines of C-programming language source code.

[1]Porting is a process of making the software to execute on a computing architecture that is different from the one for which it was originally designed

Features: Snort in just first 3 years emerged to full fledged SB-NIDS. It emerged as to have all the essential and desirable features outlined in section 4.2.4. The main features of Snort are:

- Deep Packet Inspection (DPI) which is a packet analysis involving each and every byte of packet analysis using attack signatures specified in Snort attack rules.
- Protocol analysis from layer-2 to layer-7 protocols (OSI model) including port scanning.
- Stateful Packet Inspection (SPI) which involves keeping the track of the state of network connections in order to detect stateful protocol attacks.
- Real-time attack alert and log facility for the analysis by network administrator.

4.3.2 Snort Architecture

Snort core component is the *Packet Sniffer*, the other components were added on top of the core component as plug-ins to process the sniffed packet. These components are the *Preprocessor*, the *Detection Engine* and the *Decision Engine* added as plug-ins. Figure 4.3 is the Snort architecture showing packet processing flow in plug-ins.

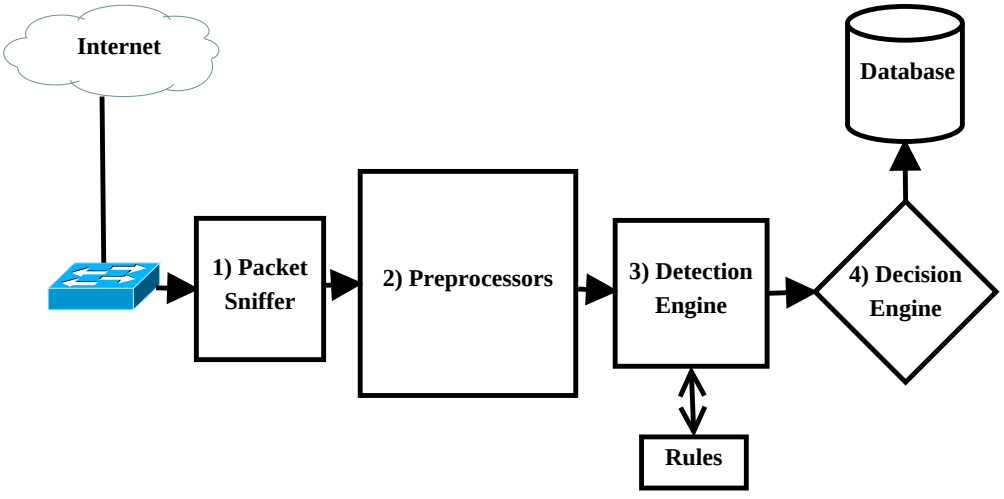


FIGURE 4.3: Snort architecture showing packet processing flow

Snort sniffed the packet through sniffer component and then performs the packet analysis in stages. First the sniffed packet protocol is decoded and the data is save to the Snort application memory area. This data is then analyse by the Preprocessor which check it for protocol based attacks. Once this complete this data is then check for attack signatures using Snort attack rules in the Detection Engine component which also

generates the detection results. This result is process by the Decision Engine component which also raises the alarm and logs the detection result.

Packet Sniffer and Decoder: A Packet Sniffer allows to tap into the data network in order to capture packets for troubleshooting and analysis. It captures them from the hardware (network interface card) in promiscuous mode using the third party packet capture library called *libpcap*. If packet sniffing is perform on the Internet facing network then it is most probably the IP traffic encapsulating many different higher-layer protocols (TCP, UDP, ICMP, OSPF etc). Packet Sniffer also decodes packet protocols with the help of its *Packet Decoder* component. The decoding results in arrangement of packet protocol data separately in Snort run-time memory so packet can analyse easily. Figure 4.4 shows the Packet Sniffer function.

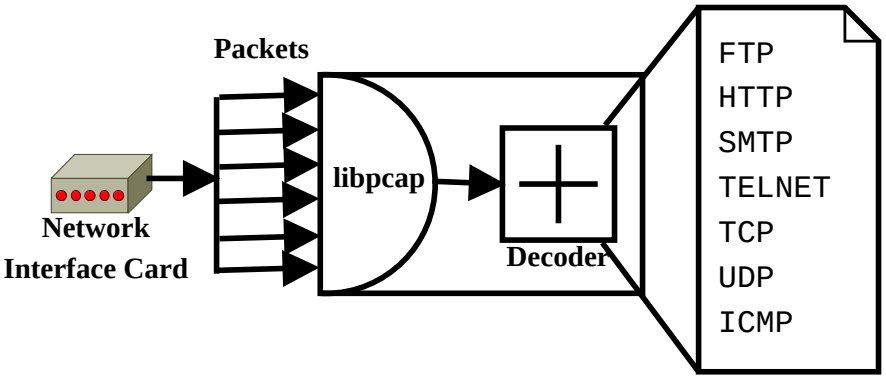


FIGURE 4.4: Packet Sniffer function

Preprocessor: Preprocessor component has a plug-in architecture covering many protocols from layer-3 to layer-7 (OSI layers). The plug-in architecture is a very useful function as it allows enable/disable the Preprocessor through the simple text file interface. For example each Preprocessor is designated to analyse specific protocol and if certain protocol requires exclusion then this can be disabled through a simple command in text file. Preprocessor performes detail protocol analysis to detect protocol based attacks in network packets (Section 2.2.1). Data normalisation is also carries out for certain protocol such as HTTP in order to make the data refined for quick signature searching in Detection Engine component. Figure 4.5 shows the packet processing flow through Snort Preprocessors.

Packet Decoder decodes the packet protocols and arranges all the packet protocol data separately in Snort run-time memory. Now the IP defragmentation Preprocessor reassembles fragmented packets and also look for any malicious fragmentation purposely done hackers to launch DoS attack (Section 2.2.1). The SPI Preprocessor then verifies if the TCP packets are a part of an established connection and if not then packet is treated as malicious. It also reassembles the packet to detect attacks that spans to

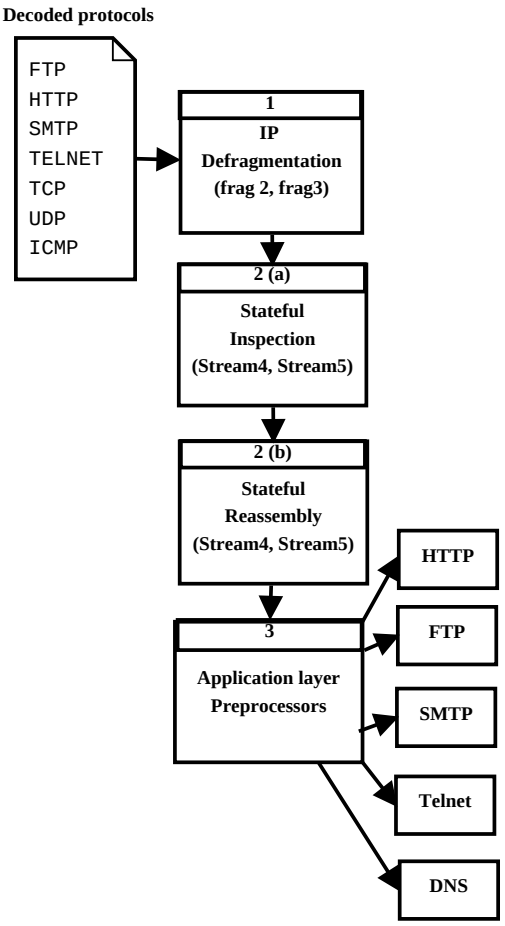


FIGURE 4.5: Packet processing flow through Preprocessors

multiple packet or extends to the sequence of packet exchanges. The collection of application layer Preprocessors normalises protocol data and also checks misuse of protocols by hackers for launching attacks. Finally, packet data passes on to the Snort Detection Engine component for further analysis.

Detection Engine: Detection Engine is the brain of Snort. This is where packet data are searched for attack signatures specified in the set of Snort attack rules and makes this brain functional. So effectively Snort search the packet data through the set of rules. If the rules match the packet data then Decision Engine takes the action. Figure 4.6 shows the flowchart of packet analysis in Detection Engine using attack rules.

Snort rules: Without rules there is no purpose of Detection Engine and hence no signature detection. So what exactly the attack rules purpose in Snort Detection Engine? The simple answer of the exact question is it is an instruction to Detection Engine. But what those instructions are? Now consider the way human language is used to describe everyday acts and instructions, this will help to understand the concepts behind Snort rules. Consider the instruction,

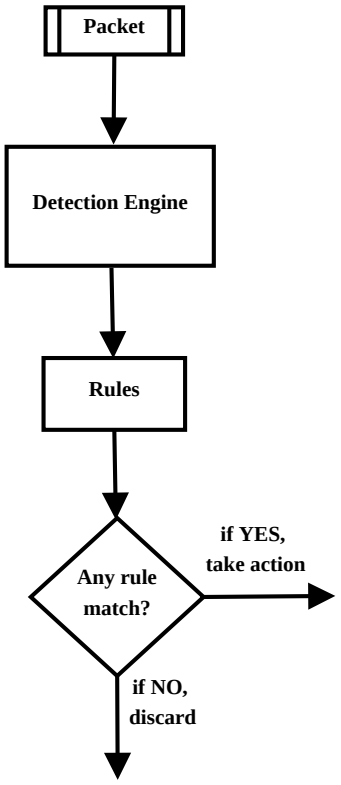


FIGURE 4.6: Packet checking in Detection Engine

"*If the landlord elder son turns up to listen the complaint then make sure to give him the exact information and request for an immediate action*".

This instruction contains a state, and then an action to perform if the state is true. Like "complaint" and "immediate action request" can only be made to the "landlord elder son". Snort rules are also this type of instructions but more specific and exact instructions supported by Snort rule language. Figure 4.7 is the Snort rule of CGI-PHF attack specified in Snort rule language syntax.

alert tcp any any → 192.168.1.0/24 80 (content: "cgi-bin/phf"; offset: 3; depth: 22; msg:"CGI-PHF attack";)

FIGURE 4.7: Snort rule of CGI-PHF attack

This Snort rule clearly specifies state and the required action if the state is true. It gives the instructions to the Detection Engine that if it encounter any TCP packet header originating from any valid source IP address/port destined to IP address 192.168.1.0/24 and at port 80, and packet payload content string *cgi-bin/phf* is present anywhere between byte number-2 to byte number-25 then take an action on a packet by alerting the administrator with a message of "CGI-PHF attack". The packet payload content for malicious pattern is searched using pattern matching algorithm. All Snort rules specifies the instruction in a similar manner. Logically they are divided into two sections: *Rule*

header and *Rule option*. Rule contents up to the first parenthesis belong to the rule header and the contents of the parenthesis belong to the rule option.

Rule Headers: It specifies seven items as shown in figure 4.8. First item from the left is *action*. There are five default actions in Snort:- alert, log, pass, activate and dynamic. The 5 rule actions are available when Snort runs in Intrusion Detection Mode. Other rule options are also available when Snort is configured to operate inline (Intrusion Prevention mode). Next to action is *protocol*. Snort currently inspects TCP, UDP, IP and ICMP protocols for suspicious activity and in the future protocols such as IGRP, GRE, ARP, IPX, and RIP will be supported. Snort's coverage of network attacks will be extended at the cost of further increases in the Detection Engine's (See section 5.3.1) computational overheads. The remaining portion of a Snort rule deals with packet IP address, port and packet flow direction information for a given rule. The first item after protocol indicates the source IP address and after this the next item is the source port. In the above example Snort rule (figure 4.7), keyword *Any* is used to specify any valid IP address and port. Specific valid IP address and port ranges for destination system just next to –> operator can also be specifies, this makes it possible to easily customise the Snort rules for a particular network providing or consuming specific internet services identified by IP addresses and ports. Next the direction operator –> indicates traffic flow from source to destination, or another operator <> indicates bi-directional traffic flow. There is no <– operator in the Snort rule description language. The item after the direction operator indicates the destination IP address and port number, which can be specified to match any address/port or specific single or ranges of both IP address and port numbers. In summary, any incoming packet that matches the rule header items (IP, Port and Direction) is selected to be analysed against the rule options. These rule header items collectively called the rule selection criterion.

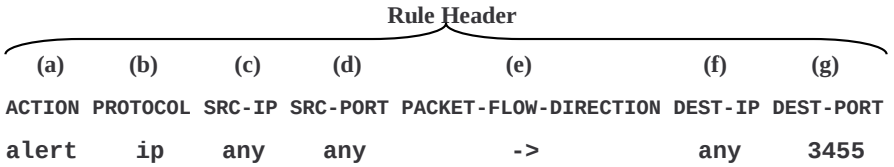


FIGURE 4.8: Snort rule header

Rule Options: Rule options define the structure of a malicious packet that include packet header and payload. These options are made up of option/value pair like in Hypertext Markup Language (HTML) options/tags. The values mentioned with these options are actually checked against the packet. When all values found in a packet then it is declared a malicious packet.

Rule option types: There are two types of rule options: header options (non-payload options) and payload options. Some header options are *Time to Live* (TTL), *Type of*

Service (TOS). These two options specify numeric values which are searched in packet headers. Two important and mostly defined payload options are *Content* and *URI-Content*. The values of these options are attack patterns comprise of strings. Payload options are search at arbitrary or defined positions in packet payload content.

Content Modifier options: Content modifier options define where and how many bytes to look into packet for attack pattern defined with the *Content* and *URIContent* options. Some main modifier options are summarised in table 4.1.

TABLE 4.1: Modifier Keywords

Modifier Keyword	Description
Depth	The *depth* keyword specify how far into a packet Snort should search for the specified pattern
Offset	The textitoffset keyword specify where to start searching for a pattern within a packet.
Distance	The *distance* keyword specify how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of the previous pattern match.
Within	The *within* keyword is a content modifier that makes sure that at most *N* bytes are between pattern matches using the Content. It is designed to be used in conjunction with the *distance* rule option.

Decision Engine: This component is the exit point for packet data that entered for processing through Packet Sniffer component. The component purpose is to take action in case of rule matches on a packet specified in Snort rule header. For example: In figure 4.7 this rule specifies that in case of rule match alert is sent to network administrator. It depends on Decision Engine configuration how the alert would be sent. It can be configured to send alert through a network connection or UNIX socket or can be stored in an SQL database server such as MYSQL or simply log the alert to hardisk.

4.3.3 Prototyping Challenges

Porting such a complex SB-NIDS software package poses several challenges. These challenges need to be met so that the objectives are achieved. These challenges are now discussed briefly:

Analysis

The three types of network packet analysis in Snort carries out in real-time are: Signature analysis, SPI and protocol analysis. All of these tasks require sufficient packet processing facility and amount of memory to make sure it should be drop free packet analysis.

Speed

Snort is a bottleneck when executes on a general purpose processor [2]. This is one of the most challenging tasks and requires detail analysis of packet analysis process as well as careful consideration of selecting appropriate processing technology.

Memory

Snort primary analysis technique is attack signature checking of known attacks in every network packet appearing on a network for the possible signs of intrusion and attacks. The signature database is in thousands in numbers and that occupy significantly large chunk of SB-NIDS runtime memory. If the Snort decides to be ported on an embedded processing architecture that has limited memory then this problem needs tackling by considering a suitable data structure of compactly storing the signature without compromising on packet analysis speed.

4.3.4 Prototyping Requirements

To meet the SB-NIDS prototyping challenges (Section 4.3.3) an appropriate development tools and processing platform is chosen which is now discussed.

Processing Platform

Speed and *Storage* are two major prototyping challenges that can be meet by choosing the right processing technology which should be ideal for network packet processing. Following processing technology are identified for prototyping:

- Network Processing Unit (NPU)
- Cluster of PC's on general purpose processor
- Graphical Processor Unit (GPU)
- Hybrid Hardware-Software (FPGA/Processor) embedded processing platform

Out of these four processing technology hybrid hardware-software embedded processing platform is more viable for SB-NIDS prototyping due to the following characteristics which has clear advantages over other three technologies:

- It has a dedicated processing unit tightly coupled to the network interface which are neither available with cluster of PC's nor with GPU unit.
- Custom hardware accelerator for offloading computationally demanding SB-NIDS sub-tasks from CPU to FPGA which are not available in NPUs.
- Multiple processing cores for parallel processing.
- Single cycle access of high speed on-chip FPGA memory for storing most frequently access data.

The processing power of FPGA is enormous in comparison to processors. It also supports instruction pipelining, parallel processing and bit-level computing which are not supported on general purpose processors instruction sets. It also has the ability to quickly reprogram and consider as shorter time to develop application than Application Specific Integrated Circuit (ASIC). Following section has a detail explanation of hybrid hardware-software embedded processing platform for SB-NIDS prototyping.

Hybrid Hardware-Software Embedded Processing Platform

Hybrid hardware-software embedded processing platform is not only available at affordable cost but it has adequate processing power, ease of hybrid hardware-software development interface and also scalable in terms of processing resources. Hybrid HandelC/MicroBlaze based embedded processing platform is also of this kind which is available by the manufacturer Mentor Graphics (Formerly Celoxica) that provide all essential features of hybrid hardware-software embedded processing platform outlined in section 4.3.4.

HandelC/MicroBlaze Hybrid Processing Technology: The HandelC/MicroBlaze based hybrid processing platform is shown in figure 4.9.

This platform is available on Celoxica RC series board *(RC300)* [109]. The main components of the board are Xilinx XC2V6000 -4 Virtex-II FPGA, 2 Gigabit Ethernet interfaces, 4 banks of 8 MB ZBT SRAM and 1 bank of 128 MB DRAM. Clearly, it is important to note that the FPGA device used in this study is only a VirtexII whereas Virtex7 device families are the current state of the art for Xilinx products. Virtex7 devices offer significantly higher clock rates and enhanced non-configurable on-chip functional units that offer the potential of corresponding improvements in performance merely by pushing the design through the Xilinx and Mentor Graphics synthesis tools for a Virtex7 device target.

MicroBlaze Soft-core processor: MicroBlaze is a 32 bit RISC based architecture optimised for Xilinx devices. It is a soft-core processor and is therefore implemented

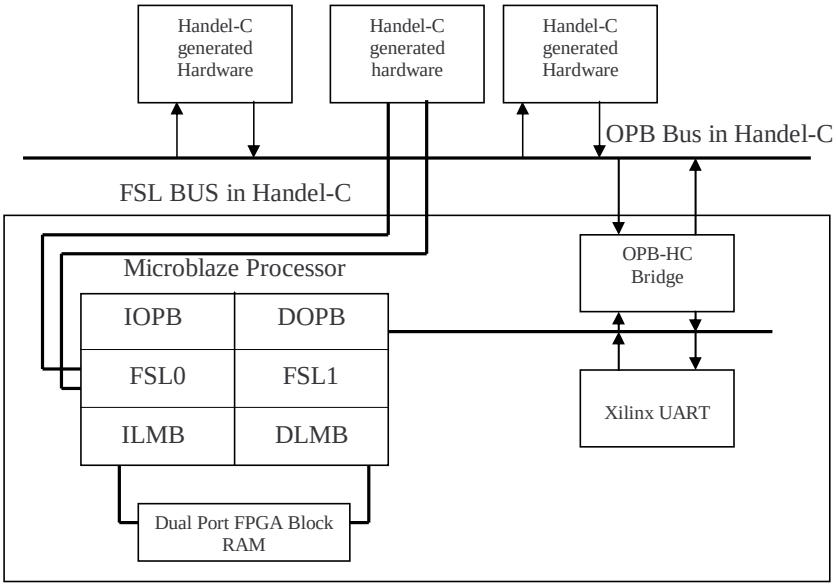


FIGURE 4.9: HandelC and MicroBlaze design system

entirely using FPGA logic resources. MicroBlaze programs are developed using the C-language compiled with a Xilinx port of the GCC compiler called MicroBlaze-gcc *(mb-gcc)*. MicroBlaze also supports instruction and data caches. These two caches are maintained in FPGA block RAM. MicroBlaze also supports three bus interfaces (FSL, OPB and LMB). These busses are Local Memory Bus (LMB), Fast Simplex Link (FSL), and On-chip Peripheral Bus (OPB).

LMB is a dedicated and low latency (memory mapped) addressable bus. In most cases small sizes of memory modules (Scratch or Cache memory) are attached to this bus. Some processors also allow custom hardware units to attach with this bus. In this system, LMB provides a link to dedicated local on-chip memory (FPGA BRAM).

OPB is a shared MicroBlaze bus which is a part of IBM's CoreConnect specification. Due to the shared nature of the OPB, reading and writing into peripheral usually takes around 10 or 11 cycles [110].

MicroBlaze Fast Simplex Link (FSL) provides a point-to-point interconnect to co-processing unit. MicroBlaze supports up to 8 FSL links that provide flexibility to attach up to 8 co-processing units. These co-processing units can be either another MicroBlaze core or a custom hardware accelerator. Dedicated instructions are also available to write into and read from this interface. The two macros for reading and writing via FSL are *getfsl()* and *putfsl()*. Reading and writing via FSL takes only up to 2 cycles per word (32-bits) on average [111].

Hardware Development: Custom hardware design on this platform can be implemented using standard RTL flows, or they can utilise Mentor Graphics hardware description language *HandelC*. Celoxica's Integrated Development Environment (IDE) called *DK* can compile HandelC based designs into EDIF or VHDL/Verilog. The Celoxica board-support libraries for the RC300 enable fast design prototyping. The key to achieving higher performance is to offload the functionality of computationally intensive NIDS program components from CPU to HandelC specified hardware accelerator which can be attached to MicroBlaze via FSL or OPB bus interfaces.

OPB provides a memory mapped interface to peripheral components. For example the address space occupied by the HandelC interface on OPB is between 0xF0000000 and 0xFFFFFFFF. This address space is used for all OPB slaves created in the HandelC design. This address space is specified in MicroBlaze Hardware Specification file *system.mhs* as shown in figure 4.10.

```
BEGIN opb_hcbridge
PARAMETER INSTANCE = opb_hc_bridge
PARAMETER HW_VER = 1.00.a
PARAMETER C_BASEADDR = 0xF0000000
PARAMETER C_HIGHADDR = 0xFFFFFFFF
BUS_INTERFACE SOPB = opb_bus
END
```

FIGURE 4.10: OPB slave memory space in system.mhs file

The address space between 0xF0000000 and 0xFFFFFFFF is also defined in *shared.h* file can be created manually. This file is shared between hardware and software design. This address space is used to split up the address space for different OPB slaves defined in HandelC.

To attach HandelC hardware accelerator on OPB bus, HandelC source file should implement read and write callback macro for hardware-software communication or message passing and another macro for defining the processing functionality of the hardware accelerator. This source file should also include opb_slave_tl.hch header file which specifies all the macro procedures required to implement hardware accelerator on OPB bus. Following are the required information the HandelC hardware accelerator source code file should contain,

- a data structure for the HandelC hardware accelerator
- a macro procedure to define functionality of the HandelC hardware accelerator
- a macro procedure to run the HandelC hardware accelerator
- a callback macro procedure to write data to the HandelC hardware accelerator

- a callback macro procedure to read data from the HandelC hardware accelerator

The data structure of HandelC hardware accelerator may consists of registers for exchanging data between accelerator and program on MicroBlaze. The four HandelC macro procedures provide the hardware accelerator functionality. Two macro procedures are callback macro for reading and writing data between hardware accelerator and program on MicroBlaze. Another macro for running the hardware accelerator in parallel with other HandelC hardware accelerator in the same design. One other macro contains the core logic for HandelC hardware accelerator.

Unlike OPB, FSL bus is not memory mapped. FSL bus is a direct link to MicroBlaze. FSL needs to be explicitly declared in MicroBlaze Hardware Specification file *system.mhs* as shown in figure 4.11.

```
BEGIN microblaze
PARAMETER INSTANCE = mblaze
PARAMETER HW_VER = 2.00.a
PARAMETER C_USE_BARREL = 1
BUS_INTERFACE ILMB = i_lmb
BUS_INTERFACE DLMB = d_lmb
BUS_INTERFACE IOPB = opb_bus

#FSL Bus description
PARAMETER C_FSL_LINKS = 1
PORT FSL0_S_READ = fsl0_s_read
PORT FSL0_S_DATA = fsl0_s_data
PORT FSL0_S_CONTROL = fsl0_s_control
PORT FSL0_S_EXISTS = fsl0_s_exists
PORT FSL0_S_WRITE = fsl0_s_write
PORT FSL0_M_DATA = fsl0_m_data
PORT FSL0_M_CONTROL = fsl0_m_control
PORT FSL0_M_FULL = fsl0_m_full
END
```

FIGURE 4.11: OPB slave memory space in system.mhs file

To attach HandelC hardware accelerator to FSL bus, HandelC source file should implement read and write callback macro for hardware-software communication or message passing and another macro for defining the processing functionality of the hardware accelerator. This source file should also include fsl_t1.hch header file which specifies all the macro procedures required to implement hardware accelerator on OPB bus. Following are the required information the HandelC hardware accelerator source code file should contain,

- a data structure for the HandelC hardware accelerator
- a macro procedure to run the HandelC hardware accelerator

- a callback macro procedure to write data to the HandelC hardware accelerator
- a callback macro procedure to read data from the HandelC hardware accelerator

The data structure of HandelC hardware accelerator on FSL may consists of registers for exchanging data between accelerator and program on MicroBlaze. The three macro procedure provides the functionality of hardware accelerator. Two macro procedures are callback macro for reading and writing data between hardware accelerator and program on MicroBlaze. Another macro for running the hardware functional unit in parallel with other HandelC hardware accelerators in design. The core logic for HandelC hardware accelerator can be provided separately in another macro procedure.

4.4 Chapter Summary

This chapter began by looking SB-NIDS architecture and features which helped to estimates the requirements of prototyping an optimised SB-NIDS. This is followed by in-depth discussion of Snort SB-NIDS software package which is a system chosen to prototype a SB-NIDS. Snort internal architecture and features are discussed in detail which helped to understand the requirements of processing and computing platform for SB-NIDS prototyping and optimisation. In the end hybrid hardware-software embedded processing platform that is chosen to developed the SB-NIDS prototype is discussed which would enable to develop an improved a scalable SB-NIDS prototype solution. In the following chapters the series of algorithms and hardware architectures are discussed as part of the SB-NIDS prototype development and optimisation.

Chapter 5

Design and Implementation

Sir Frederick Henry Royce was an engineer and one of the founders of Rolls-Royce Ltd said:

> "Strive for perfection in everything you do. Take the best that exists and make it better. When it does not exist, design it."

This engineering philosophy is followed in the development of Signature based Network Intrusion Detection System (SB-NIDS) prototype. Strive for perfection is the core principle of engineering and is also applied in SB-NIDS prototyping. This is practically achieved by choosing the best available hardware and software tools such as hybrid hardware-software processing platform and Snort SB-NIDS software package. There are few cores SB-NIDS features that are necessary and needs development to make SB-NIDS packet analysis speed better. These features are carefully designed and implemented by considering the development challenges outlined in section 4.3.3.

5.1 Chapter Roadmap

The rest of the chapter is outlined as follows:

- In section 5.2, MMU-Snort I or SB-NIDS prototype development using hybrid HandelC-MicroBlaze embedded processing platform is presented. This involve a brief analysis of Snort SB-NIDS software execution to understand SB-NIDS processing requirement, Snort SB-NIDS software architecture restructuring and mapping details to processing platform.

- In section 5.3, MMU-Snort II or Pattern Matching Hardware Asccelerator (PMHA) development is presented. This involve an analysis of Snort's Detection Engine component that performs pattern matching and an analysis of Bloom filter data structure for compactly storing large number of attack patterns. This is followed by detail PMHA design and implementation description.
- In final section 5.4, MMU-Snort III or further improvement of PMHA is presented. This include algorithmic and architectural improvement of PMHA to speed up pattern false positive pruning process and efficiently search longer patterns (> 64 bytes).

5.2 Snort Port on Hybrid Hardware-Software Processing Platform (MMU-Snort I)

This section contains detail description of MMU-Snort I development using hybrid hardware-software embedded processing platform and Snort SB-NIDS software package. This novel development platform and Snort SB-NIDS helped to come up with much improved SB-NIDS solution in a significantly shorter time with all required features (Section 4.2.4).

5.2.1 Analysis

The application execution analysis is the profiling of Snort. The profiling result of Snort (ver 2.6.1.5) on Personal Computer (PC) using a GNU gprof profiler (v 2.16) was obtained. Snort was executed on Intel 2.0 GHz processor on Debian Linux 2.6.18. It was configured with five Preprocessor components (Stream4, frag2, HTTP Inspect, Telnet decode and sfportscan) and 6565 number of rules. The network packet trace used to obtain the profiling result was MIT Lincoln Lab's 1998 DARPA Intrusion Detection Evaluation dataset (Network trace file (tcpdump format)) [107]. Table 5.1 shows the percentage of the total execution time used by each component[1].

TABLE 5.1: Profile of Snort on PC

Component	**Execution %**
Detection Engine	49%
Packet Decoder	19%
Preprocessors	21%
Decision Engine	11%

[1]Profiling results depend on Snort configuration and test data.

The Detection Engine component consumes the highest number of Central Processing Unit (CPU) cycles. In this component most of the CPU time is spent on testing the condition of rules (Protocol and packet content). It was observed that 35% of overall execution time is spent in the Detection Engine method *(acsmSearch2())*. This method is an implementation of the Aho-Corasick [55] pattern matching algorithm. In the Preprocessor component the bulk of the CPU computational time is spent in preparing the packet for evaluation. These computations involve frequent memory accesses such as tasks related to fragmented packet assembly, storage and retrieval of TCP connection information (IP addresses, sequence number, port numbers) from memory and modification of HTTP packet content for further inspection. In the Packet Decoder component, the bulk of the CPU time is spent in extracting data from the packet for further inspection. Decoding also involves computations such as bit manipulation for the Packet Checksum calculation. Logically simple bit manipulation operations are typically inefficiently supported by general purpose processor instruction sets and such operations must be mapped onto a series of shift and mask instructions. Decision Engine performance is entirely dependent on the way the detection result is processed. If the Decision Engine stores results to a database/disk for later analysis then it may consume a high number of CPU cycles to access and write detection results to databases/disk. In this experiment Snort was configured to send alerts to the console.

5.2.2 Design

Snort on RC300 development board is ported by mapping Snort components on HandelC-MicroBlaze based environment. This mapping involve restructuring the Snort architecture which is carried out by considering the followings point:-

1. The modified version of Snort for hybrid HandelC-MicroBlaze based processing platform and the standard software distribution of Snort should produce the same detection results.

2. The modified version should also be easily customised via the same simple interface as the original Snort distribution in which features can be easily added/removed via a simple text file.

3. Snort dependencies on external software libraries/Operating System (OS) features should be removed in order that there are no difficulties in porting to embedded target that for example does not possess libpcap for packet capture and that is not running any OS.

4. Gain efficiency by offloading processing load from processor to Field Programmable Gate Array (FPGA) hardware without consuming too much FPGA area/resources. Careful design is required to avoid the inadvertent introduction of performance bottlenecks due to communication between hardware and software components.

All these points were considered carefully for porting along with application execution analysis result. The main goal of the design and implementation was set to successfully execute the Snort on new HandelC-MicroBlaze environment (Point 1 above). File based customisation of Snort was achieved easily with the help of MicroBlaze memory file system (Point 2 above). The last two points (above) are significantly hard and required lot of time and efforts because they involved careful design and implementation. Following changes were proposed in Snort:-

- Remove Snort Packet Sniffer component reliance on 3^{rd} party packet capture library (libpcap) because the OS call trap and return overhead, coupled with the buffer copies of packets contribute to unnecessary overhead in Snort systems. This was done by implementing a packet capture HandelC hardware accelerator that captures raw packets from gigabit Ethernet interface tightly coupled to the FPGA and the MicroBlaze providing high speed and low latency access for packet capturing.
- Snort core components the Detection Engine and the Preprocessor decided to port initially on MicroBlaze core by removing their dependency on external libraries (arpa, socket, pcre)[2]. This porting decision was based on the idea that in future MicroBlaze cores can be used to deploy critical Preprocessor component on individual processor and Detection Engine component which executes pattern matching to offload from MicroBlaze to FPGA.
- Snort's Decision Engine functionality is reduced to alert the administrator if packet matches any rules. This was achieved by offloading Decision Engine from MicroBlaze to FPGA which sends the alert through gigabit Ethernet interface. A separate application would be developed in future to provide the logging/saving of the detection results to the database or disk and to present the result in a more interactive Graphical User Interface (GUI).

The design of the modified Snort ported on RC300 board is shown in figure 5.1. A HandelC Packet Capture Hardware Accelerator (PCHA) directly captures packets without any delay from gigabit Ethernet interface-0 and stores it in Static Random Access Memory (SRAM) BANK3 for further inspection. The gigabit Ethernet interface is tightly

[2]the current prototype does not support regular expression *(pcre)*

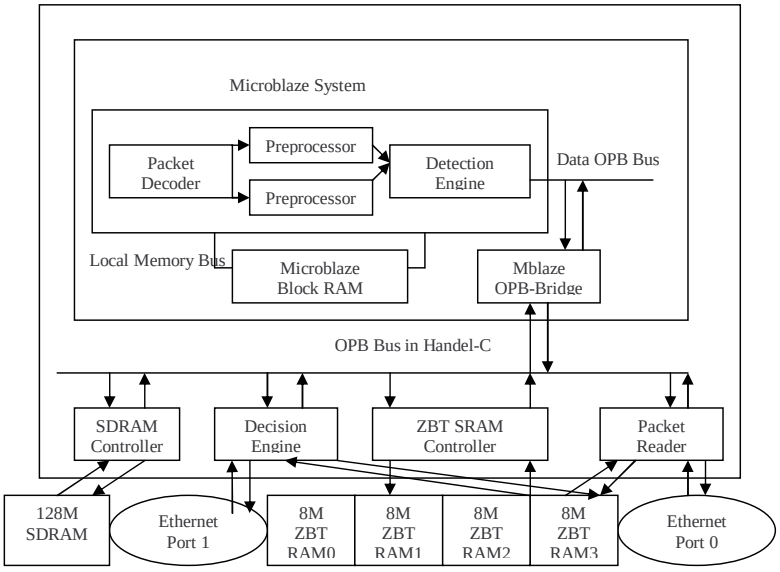


FIGURE 5.1: Snort on RC300 board

coupled to the FPGA and the MicroBlaze providing high speed and low latency access for packet capturing. The hardware implementation of packet capturing is a significant improvement over the use of a software-based libpcap library as it does not require any operating system interrupt calls, nor the copying of data from kernel to application buffers. This design can still be further improved by creating a ring buffer for received packets in SRAM/SDRAM (Static Dynamic Random Access Memory) or altering the design to attach the PCHA via the high-speed FSL bus in order to provide a direct data transfer link between packet capturing custom hardware and processor.

Snort's three components with major roles in packet analysis are the Packet Sniffer, Preprocessor and Detection Engine are ported on MicroBlaze. These components are collectively called the *Core Engine*. The Core Engine is the heart of this SB-NIDS and it involves the computationally significant operations such as pattern matching, packet classification and stateful inspection that are the operations for further optimisation ideally using FPGA logic resources or parallel processing of multiple processor cores.

In the original distribution of Snort, detection results produce by the Decision Engine after packet analysis are typically log or store in database systems, displayed on a console or sent over a network. In this system detection results are written first to SRAM BANK3. This is then read by Decision Engine Hardware Accelerator (DEHA) that sends the detection result over a network via gigabit Ethernet interface-I. A separate application is decided to be developed in future that will use this result to log/store remotely which would enable the detection result available for later analysis by Network Administrators.

5.2.3 Implementation

Snort porting on hybrid HandelC-MicroBlaze based embedded processing environment required source code level changes in Snort application. These changes include removing methods not required on processing platform and providing implementations for functions/methods that are not available in MicroBlaze-C *(mb-gcc)* library. Some of these methods are Internet address manipulation methods (htonl(), ntohs() etc.) and socket library methods (accept(), connect() etc.). Apart from these changes the two new hardware accelerators are added to Snort port for optimising Snort's packet analysis performance. The two hardware accelerators are: *Packet Capture* and *Decision Engine*. Packet capturing facility is a part of Snort sniffer component in original Snort package which in this prototype offloaded to hardware for bottleneck free packet capture (Section 4.3.2). Decision Engine in original Snort package is also a separate Snort component which also offloaded to FPGA. These two hardware accelerators are attached to MicroBlaze On-chip peripheral bus (OPB). The communication between these hardware accelerators on OPB bus and rest of the Snort core engine on MicroBlaze is facilitated with the help of HandelC library (opb_slave_t1.hcl) methods which has a code for controlling OPB bus and enabling communication between hardware accelerators and software on MicroBlaze.

Packet Capture Hardware Accelerator (PCHA)

PCHA is implemented using the necessary data structure and macro procedures (Section 4.3.4). Its data structure consists of two 1-bit status registers (busy and start register) and a 32 bit packet length register to store captured packet size in bytes. A packet length register along with status register initially set to clear or zero. Figure 5.2 shows the PCHA architecture.

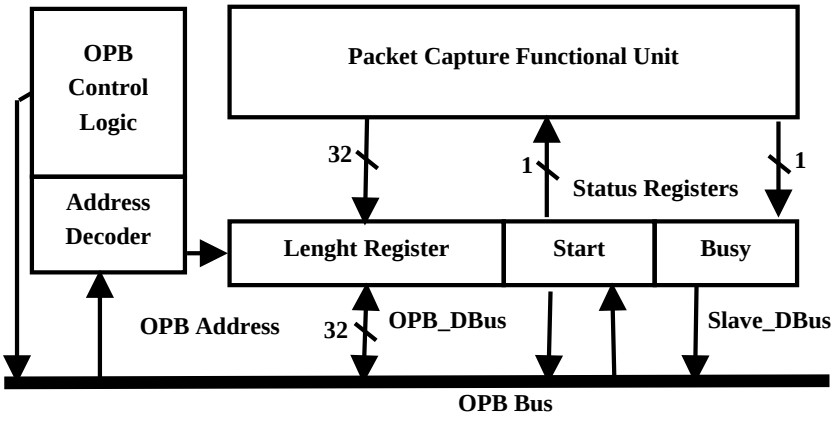


FIGURE 5.2: Packet Capture Hardware Accelerator (PCHA) architecture

The communication between PCHA and rest of the Snort port on MicroBlaze is synchronised and supported by HandelC status registers, and HandelC methods and macro procedures. Snort on MicroBlaze calls the *OPBWriteUINT32(START_REG_BASEADDR)* register write method with start register base address as a parameter. A callback write macro procedure *OPBEthernetWrite()* of PCHA activated which then writes the contents of the OPB data bus to the start register. This status register update signals the core packet capture logic in *OPBEthernet()* to capture the packets from RC300 board gigabit Ethernet interface-0. It first sets the busy status flag to 1 and begins the packet capture. The complete packet capturing time depends mainly on the packet length. It takes 1-clock cycle for reading 1-byte of packet data from Ethernet interface and another 2-clock cycle to store every 4-byte of packet data to store in SRAM BANK-0. Once the packet completely captured, the macro procedure writes the packet length to the length register and clears the busy status register to zero.

During packet capture in accelerator hardware, Snort on MicroBlaze read the busy status register on every clock cycles using a *OPBReadUINT32(BUSY_REG_BASEADDR)* register read method with busy register base address as a parameter. This triggers a callback read macro procedure *OPBEthernetRead()* in HandelC PCHA that reads the contents of the busy register and pass it through OPB bus signals to Snort on MicroBlaze. If Snort found busy flag cleared then it reads the packet length and analysis on the captured packet begins with protocol decoding.

Decision Engine Hardware Accelerator (DEHA)

HandelC DEHA is also implemented using the necessary data structure and macro procedure (Section 4.3.4). It comprises of a 1-bit busy register and 4-byte alert register to represents the number of attack rules successfully matched on a packet. Figure 5.3 shows the DEHA architecture.

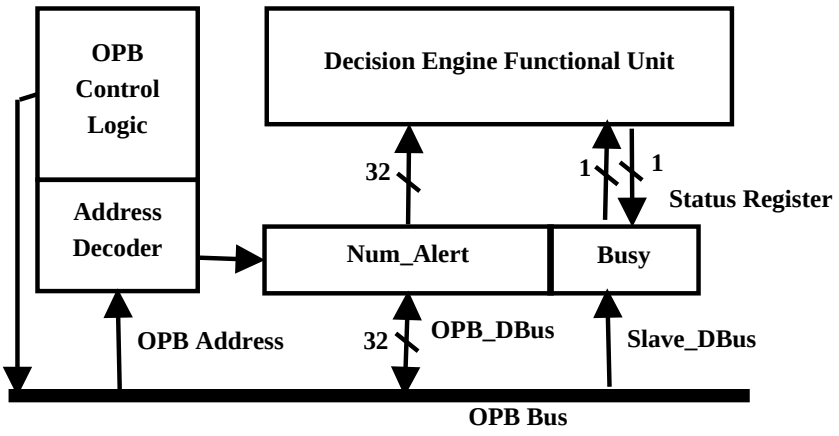


FIGURE 5.3: Decision Engine Hardware Accelerator (DEHA) architecture

DEHA functions in a very similar manner like PCHA. Busy and an alert register initially sets to 0 to indicate that DEHA is not busy and there is no attack rules alert to process. Complete functionality of Decision Engine is defined in four HandelC macro procedures.

When all Snort components on MicroBlaze finished packet analysis then detection result is wrote to SRAM BANK-3 and total number of attack rules matched on a packet wrote to alert register. This is carried out by calling a *OPBWriteUINT32-(NUMALERTS-REG-BASEADDR)* and *OPBWriteUINT32(BUSY-REG-BASEADDR)* register write method with alert and busy register base address as a parameter respectively. These methods call triggers callback write macro procedure *OPBResultWrite()* in hardware accelerator which then writes the contents of the OPB data bus to the alert and busy register respectively. At the same time the core logic in macro procedure *OPBDecisionEngine()* that runs in infinite loops checks the busy register status in every clock cycle and on finding that it's status is set to 1, it starts reading detection results from SRAM BANK-3 and sends them through gigabit Ethernet interface-1. This Ethernet interface can be connected to network administrator console or PC where a real time software application can read this detection result and displayed it on screen for analysis.

In summary, MMU-Snort I or prototype SB-NIDS design and implementation is described which is carried out by porting Snort SB-NIDS software package on hybrid HandelC-MicroBlaze based embedded processing platform. The novelty of this effort is the first ever SB-NIDS on hybrid hardware-software embedded processing platform. This SB-NIDS performance is also tested to identify any packet analysis speed performance improvement and any possible performance bottlenecks (Section 6.3). The performance bottlenecks is then improved by further research and development (Section 5.3 and Section 5.4).

5.3 Pattern Matching Hardware Accelerator (MMU-Snort II)

SB-NIDS prototype testing results shows packet analysis speed improvement and also indicated some packet processing bottlenecks (Section 6.3). One of the bottlenecks identified is the pattern matching algorithm which performs attack signature search in packet payload. Another issue is the large number of attack signatures that does not fit completely on embedded processing platform memory. To deal with two these issues and design a solution a brief analysis is carried out first.

5.3.1 Analysis

This analysis involved understanding the internal structure and working of Snort's Detection Engine component which performs pattern matching. This is followed by a discussion on Bloom filter data structure for understanding how it can make possible to compactly store large number of members or attack patterns in embedded processing platform memory for fast lookup [5].

Detection Engine Functions

Detection Engine and Snort attack rules were discusses before briefly (Section 4.3.2). Their functions are now discussed in detail.

Each and every packet in Snort is pass through the series of processing stages as shown in figure 5.4. The numbering shows the flow of packet from one processing stage to another.

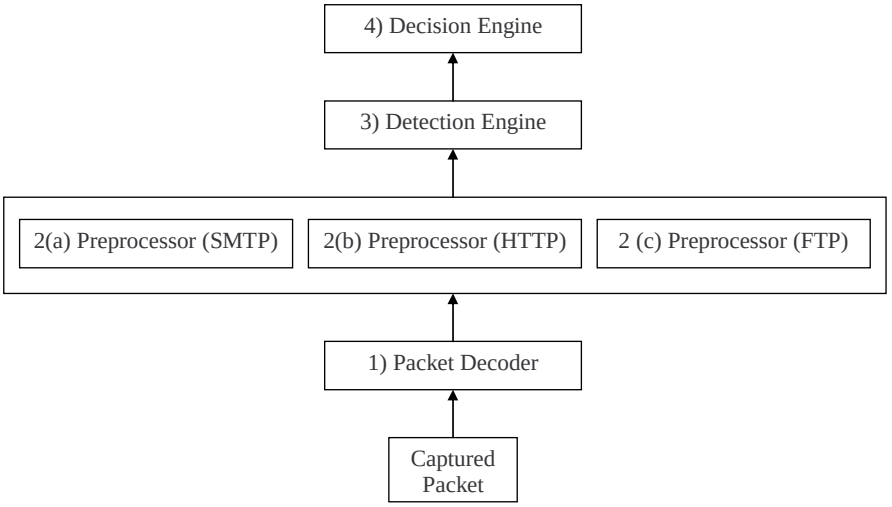


FIGURE 5.4: Key Stages of Snort

Packets are firstly captured from network interface(s), and then decoded and analysed by Preprocessor component(s). Next, packets are passed onto the Detection Engine component, where Snort attack rule are selected and evaluate on packet. In brief, the Detection Engine performed three main operations: *Rule Parsing*, *Rule Selection* and *Rule Evaluation*.

Rule Parsing

At Snort application initialisation stage, the Detection Engine component reads and parses all the rules found in a Snort configuration text file (Snort.conf) in order to

generate a data structure for rule evaluation, as depicted in figure 5.5 known as the *Snort rule tree (SRT)* [1].

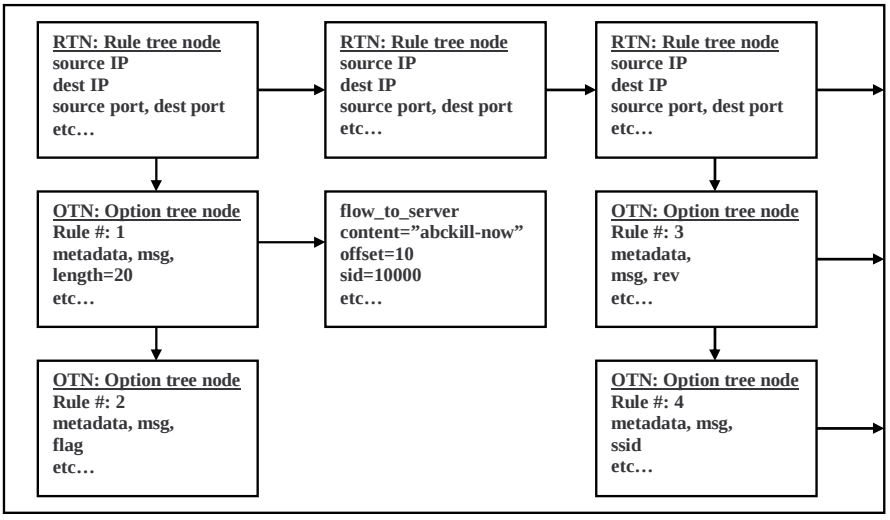


FIGURE 5.5: Parsed structure of Snort rules in memory (SRT)

SRT is made up of *Rule tree nodes (RTN)* and *Option tree nodes (OTN)*. An RTN contains the data associated with a rule header, whereas an OTN contains data associated with rule options that include rule meta-data and detection options such as offset, depth (Section 4.3.2). The SRT groups all OTNs with the same rule header under a single RTN in order to facilitate fast rule evaluation on packets.

Rule evaluation

Rule evaluation in Snort involves checking packet header for invalid protocol values and packet payload for malicious pattern. Header checking is straight forward process of checking numeric values where as payload checking involves pattern matching algorithm. Early Snort versions used Boyer-Moore [54] pattern matching algorithm with relatively small memory requirements. It is very inefficient with larger number of attack pattern search because each pattern search in packet payload occurs one-by-one. Thus, Boyer-Moore algorithm for pattern search in SB-NIDS can easily become a victim of DoS attack on high data rate network [112]. Subsequent releases of Snort versions exploited the power of Aho-Corasick based multi-pattern matching algorithms [55] which search the whole number of patterns in packets in one go using finite state machine that it creates in Snort run-time application memory (Figure 5.6).

This algorithm is a significant improvement in terms of speed over Boyer-Moore pattern matching algorithm at the cost of large amount of run-time pattern memory for finite state machine. Even finite state machine memory optimisation does not result in any

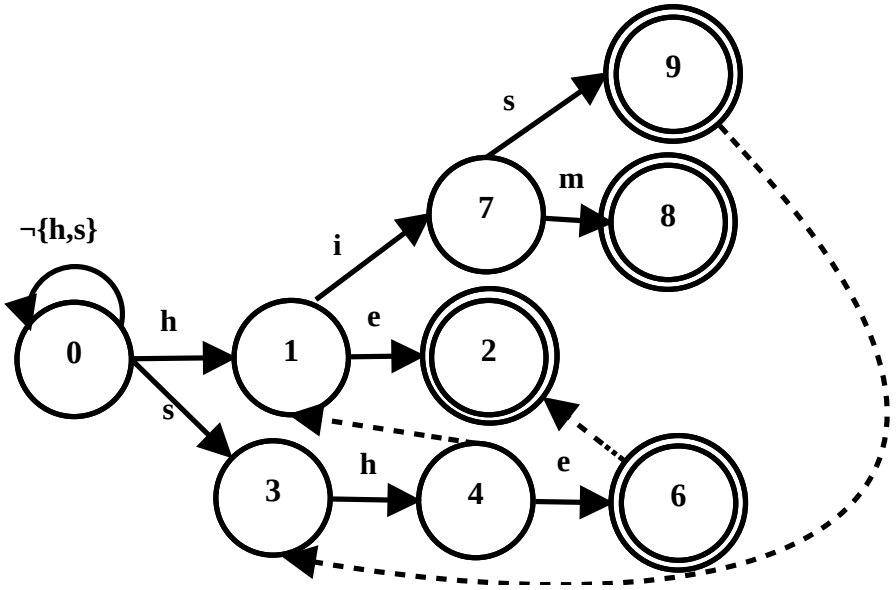


FIGURE 5.6: A state machine concept constructed using patterns "he, she, him, her, his"

significant reduction of run-time memory requirement for Aho-Corasick [97] (Figure 6.6). This same issue also caused problem to execute Snort port on embedded processing platform with all Snort attack rules (Section 6.3)). Still Aho-Corasick multi-pattern search algorithm performance is much better than Boyer-Moore single pattern search both in terms of speed and its worst-case performance. Therefore, in recent versions of Snort (such as Snort version 2.0 and after), Aho-Corasick is the default pattern matching algorithm to search packet content for attack patterns. This search is supported by SRT which helped to drive the pattern search in packet content as well as invalid protocol values search in packet header both specified in Snort attack rules. This search is called *Rule Evaluation* on packet involved packet header check and packet payload search using rules. Before the rule evaluation SRT also helps to select rules for evaluation using packet classification (Section 2.4.7) process commonly referred as *Rule Selection* in the context of SB-NIDS.

SRT also contains the Snort rule content modifier options specified for some 45% of attack rules. These options identified as crucial for pattern search performance because it reduced the overall pattern search time in packet content by specifically mentioning the number of bytes and offset to search into packet contents instead of all packet content (Section 4.3.2). For example, consider the Snort rule as shown in figure 5.7, which specifies that an alert should be sent to the network administrator if the *ttl* value of the IP packet under inspection is less than three and the pattern *Kill Now* is found in the packet payload between byte four to twenty-four. Pattern search is performed very quickly for this rule due to the *depth* and *offset* Snort rule content modifier options which specifies the exact location to search instead of all packet content. If these modifiers are

not present in rule then Detection Engine has to search all packet payload which may be around 1500-bytes (Ethernet MTU).

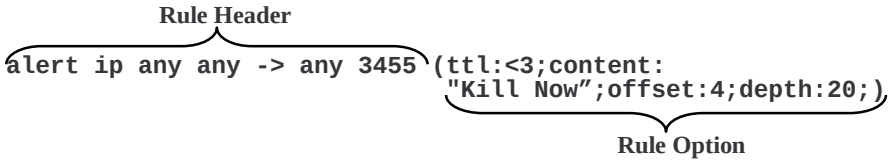


FIGURE 5.7: Example Snort rule

Bloom Filter

A Bloom filter is a space-efficient probabilistic set membership data structure. It allows set members insertion and query but does not allow deletion (deletion is possible with a *Counting Bloom Filter [113]*). An insert operation on a new set member is implemented using multiple hash functions applied to an integer value (representing the member). A character string such as "FAT" can be represented as an unsigned integer value that is created by concatenating the ASCII character codes (bit patterns), see figure 5.8. The results of applying hash functions to the integer value representing each and every keyword string are bit-wise OR-ed together in order to create the stored Bloom filter bit-pattern value. Bloom filters have been successfully applied to computer security based applications such as Email Spam Filters [114], Network Intrusion Detection Systems [81] and Computer Worm Detection Systems [115].

```
>>>> (ord('F') << 16) + (ord('A') << 8) + ord('T')
```

FIGURE 5.8: Python code: 3 character string to integer conversion

Bloom Filter Algorithm

An empty Bloom filter is a bit-array *(B = 0,...,m-1)* (also known as *bit-vector*) of m number of bits, all set to *0* initially as shown in figure 5.9. It is used to efficiently and compactly represent a set of bit-strings S, with n number of members *($S = x_1,x_2,...x_n$)*. A Bloom filter can be programmed (Insert) for a bit-string x of a set S using k number of hash functions *($h_1()...h_k()$)*. textitk hash functions are computed on any bit-string x resulting in k hash values. Each of these k hash values represent a single bit position *B[k])* set to *1* in a bit-vector of size m. Hence each hash value computed on any bit-string x is used for setting k number of bits to *1* in the size m bit-vector. Each one of the k hash values can be interpreted as an integer in the range from 2^0 to 2^{m-1}.

Figure 5.10 illustrates the Bloom filter programming. Two bit-strings x_1 and x_2 are programmed in the bit-vector B of $m = 10$ number of bits with $k = 3$ number of hash

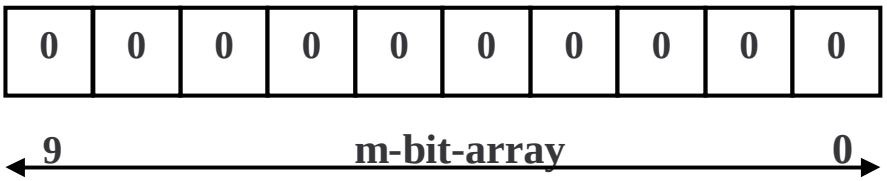


FIGURE 5.9: Empty Bloom Filter

functions. It can be noticed that two different bit-strings *(x_1 and x_2)* set the same bit positions in bit-vector index of *B[2]* corresponding to a hash value of 2^2, this is known as a hash collision.

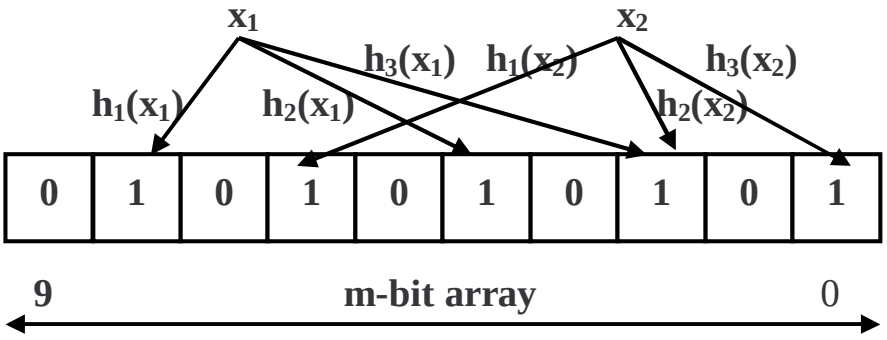


FIGURE 5.10: Insert bit-strings (x_1) and (x_2)

Querying the bit-vector B for a bit-string x is similar as insert operation. Using the same k number of hash functions $h_1()...h_k()$, k hash values are computed for any bit-string x of a set member S. The k hash values are checked against the stored Bloom filter bit position values *B[k]*. If atleast one of the k bit position *B[k]* is *0*, the member is declared to be a non-member of a set S or not programmed in a bit-vector. If all k bit positions *B[k]* in bit-vector B are found to be *1*, the member is declared to be a member of set S or found programmed in a bit-vector with certain probability. If all k bit positions *B[k]* in bit-vector B are found to be set 1 for bit-string x but it is not a member in a set S or not programmed in a Bloom filter, then it is said to be a false positive.

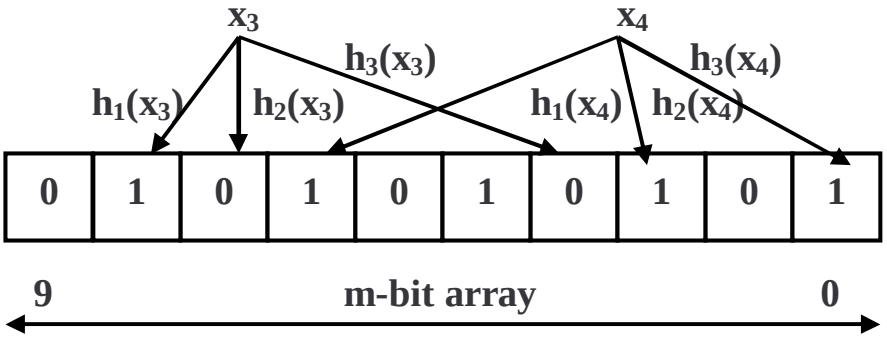


FIGURE 5.11: Query bit-strings (x_3) and (x_4)

Figure 5.11 illustrates the Bloom filter query. Membership query for bit-string x_3 shows it is not a member of a set S or not programmed in bit-vector as two k bit positions *B[3]*

and B[7] found to be 0. However, membership query for bit-string x_4 is false positive as k hash values maps to bit positions *B[6], B[2] and B[0]* set by bit-strings x_1 and x_2 of a set S (See figure 5.10). The advantages of Bloom filter is that they can efficiently search to see if a given bit-string contains one or more members of a set of keywords, unfortunately false positive results must be resolved and these require further pattern matching.

Bloom Filter Characteristics

The Bloom filter helps to overcome Aho-Corasick multi-pattern matching algorithm two main computational factors: *Memory space* and *Search time* by observing its following characteristics.

- Bloom filter membership query does not involve any bit-to-bit members matching unlike similar hashing coding techniques (Hash table). Membership tests involved checking k bits status (set to 1) in a pre-computed bit-vector. Thus providing mechanism to lookup any length or size member with constant lookup-time *O(n)*.
- Unlike other data structures (arrays, linked lists, hash tables) and popular pattern matching algorithms (Boyer-Moore and Aho-Corasick), the members of a set S are not stored in the bit-vector B, instead each member is represented only by k number of bits position in a bit-vector *B[k]* (Bits are shared between members). Thus making it suitable for compactly storing large number of members in a bit-vector B.

This space and time advantage in a Bloom filter is achieved at the cost of allowable errors. In other words, querying bit-pattern x in a bit-vector may return a false positive true result even when bit-pattern x is not in the set or not programmed in a bit-vector (Figure 5.11). This happens when k number of hash values for a bit-pattern x coincide with bit-vector B positions corresponding to the hash values set to 1. The probability of such allowable errors occurring for any bit-pattern x query is called the false positive probability. Bloom in his work [5] defined the false positive probability as,

$$f = (1 - (1 - \frac{1}{m})^{nk})^k \approx (1 - e^{-\frac{nk}{m}})^k \tag{5.1}$$

Where m is a size of a bit-vector, n is the total number of bit-patterns in a set S and k is the number of hash functions to be computed on a bit-pattern. It can be noticed from the equation that m, n, S and k all affect the false positive rate. For n number of

bit-patterns the false positive can be reduced by choosing appropriate values of m and k. The value of m needs to be large compared to the value of n, whereas, the optimal value of k achieving a low false positive rate depends on the ratio m/n, i.e. the average number of bits occupied by single member. This optimal k is calculated by minimising equation 5.1 as,

$$k = (\frac{m}{n}) \times ln2 \tag{5.2}$$

k is an integer and this value of k produce lowest possible false positive rate with respect to the values of m and n.

From this analysis it can now be concluded that the Detection Engine component of Snort port on RC300 board requires major code level changes in order to integrate the PMHA because part of Detection Engine component performs pattern matching not the whole component. Also Snort application specific knowledge (attack rule content modifier options) inclusion to PMHA can improve the pattern matching performance due to lower number of byte search. It can also be deduced from the Bloom filter discussion that its characteristics are suitable for compactly storing the attack patterns in FPGA memory as well as to support quick pattern matching in packet content. Keeping the lower false positive rate is also crucial for PMHA performance which can be lower by bit-vector lookup with higher number of hash values per member or pattern. However, this may easily affect optimisation effort due to higher number of hash value computations per pattern. This issue can be overcome by implementing hardware friendly optimal hash function as well as using a simple mathematical calculation to quickly compute large number of hash values per pattern (Section 5.3.3).

5.3.2 Design

The three core operations performed by Detection Engine are:– *Rule Parsing*, *Rule Selection* and *Rule Evaluation*. All three core operations required source code level modification in order to design, develop and integrate PMHA to come up with MMU-SnortII prototype. Rule evaluation which involved packet header and payload check with rules required major source code level modification in which the packet payload check (pattern matching) is offloaded from MicroBlaze to FPGA. Figure 5.12 is the top level diagram of modified Detection Engine component partitioned between hardware and software and communicating or passing data using MicroBlaze Fast Simplex Link (FSL) bus.

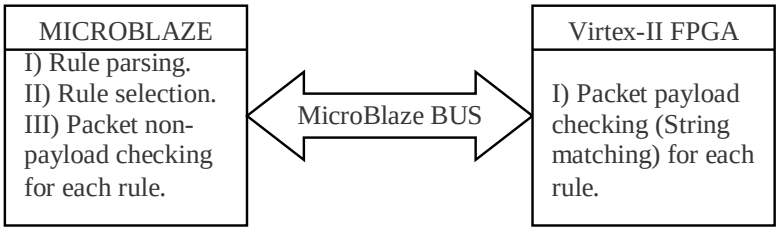


FIGURE 5.12: Top level diagram showing modified Detection Engine

Software Design

On MicroBlaze, the modified Detection Engine now performs:– rule parsing, rule selection and packet header check (Part of rule evaluation). The rule parsing is modified and it no longer creates the Aho-Corasick finite state machine. Rule selection operation remains unchanged while rule evaluation went through major modification in which packet payload check (part of rule evaluation) is offloaded from MicroBlaze to FPGA.

Hardware Design

On FPGA, packet payload search (part of rule evaluation) for attack pattern in packet payload is performed. This search is based on Bloom filter based pattern search approach. Figure 5.13 shows the block diagram of PMHA.

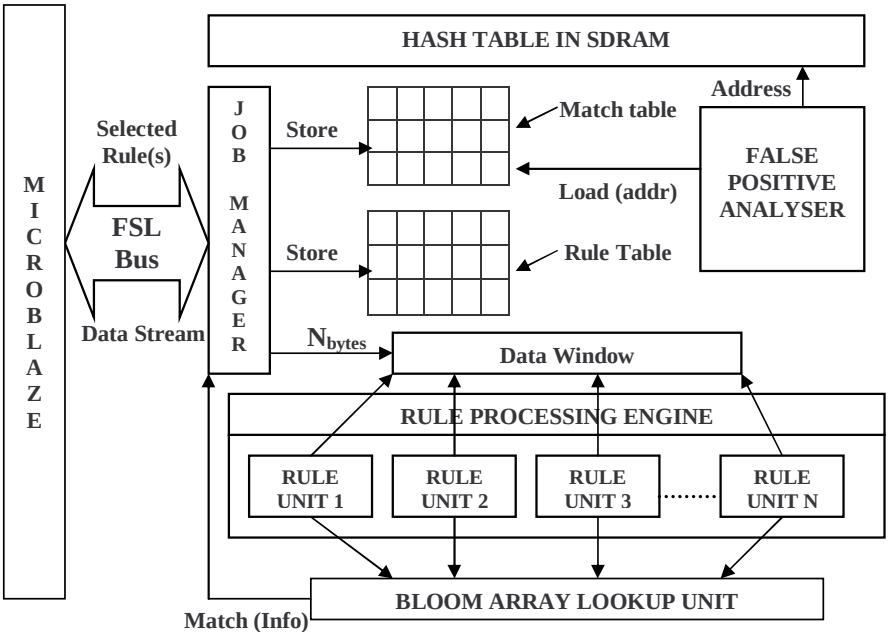


FIGURE 5.13: Block diagram of pattern matching hardware function unit

The PMHA has four main modules: *Job Manager*, *Rule Processing Engine*, *Bloom Filter Lookup Module*, *False Positive Analyser* that actively function together as pipelined

pattern matching hardware modules. This design is better and has advantages over previous Bloom filter based pattern matching hardware solutions and Snort Rule Processing System (Section 3.5.3) due to following reasons:

Application specific knowledge: It is the first time ever Snort rules content modifier options are integrated into pattern search algorithm that specifies clearly the part of packet content to search for attack patterns. This results in less computation and lower number of clock cycles due to:

- Less number of hash computation.
- Low numbers of Bloom filter lookup or block memory access.
- Few pattern for pruning in false positive analyser.

In comparison, the previous Bloom filter implementations fully search or every byte of the packet payload for locating attack patterns [81].

Hash computation technique: The conventional way of computing any number of hash values in hardware or FPGA is hash computational unit. The same hash computational unit can be used to compute multiple numbers of hash values. But for computing ten hash values or more this will cost high computation time. To fix this problem another approach is to have multiple numbers of hash computation units which will cost more FPGA logic resource and power and thus also result in performance degradation. To overcome this problem a novel mathematical technique by Kirsch [6] is used to implement the 2-to-N hash module that uses two hash values to compute another eight hash values in just 2 clock cycles without any increase in asymptotic false positive probability of Bloom filter. The advantages of implementing 2-to-N hash module in comparison to multiple hash computation units are summarised as:

- Lower FPGA logic area usage and minimal power consumption
- Large number (eight) of hash values computation in just two clock cycles

Integrated Rule Processing System: Snort Rule Processing Systems were either implemented as standalone systems on FPGAs or on Network Processors (Section 3.5.3). The standalone solutions aim were to present optimised rule processing solutions but did not clearly demonstrate how these Snort Rule Processing Systems would be integrated to function or process attack rules for SB-NIDS. This design presents an integrated Rule Processing System to snort SB-NIDS port (MMU-Snort I) using hybrid hardware-software processing platform which never done ever before and so advances the start

of art. This state of the art also supports the highest number of attack rules and compactly stores the attack pattern in FPGA local memory. Figure 5.14 shows this rule processing/evaluation on packet.

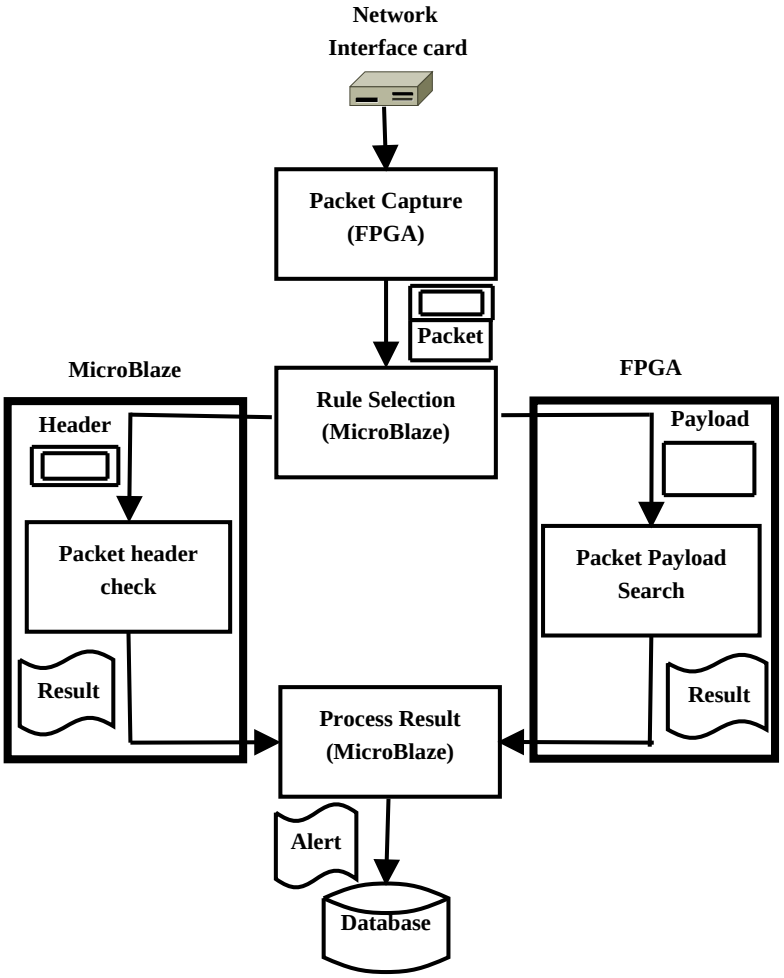


FIGURE 5.14: Packet processing flow for Snort rule evaluation

Each rectangle in the figure represent a processing component. Packet capture and Packet payload search components are HandelC hardware accelerators attached with MicroBlaze on OPB bus and FSL bus respectively. The rest of the components execute on MicroBlaze[3].

Packet is captured by the packet capture component from RC300 gigabit Ethernet port-0 and stored in off-chip SRAM BANK-0. Next the key packet header values are compared with rule header in the rule selection component on MicroBlaze that returns the subset of rules for evaluation on packet as shown in figure 5.15.

[3]Figure 5.14 highlights only rule evaluation in Detection Engine and does not show other Snort processing component

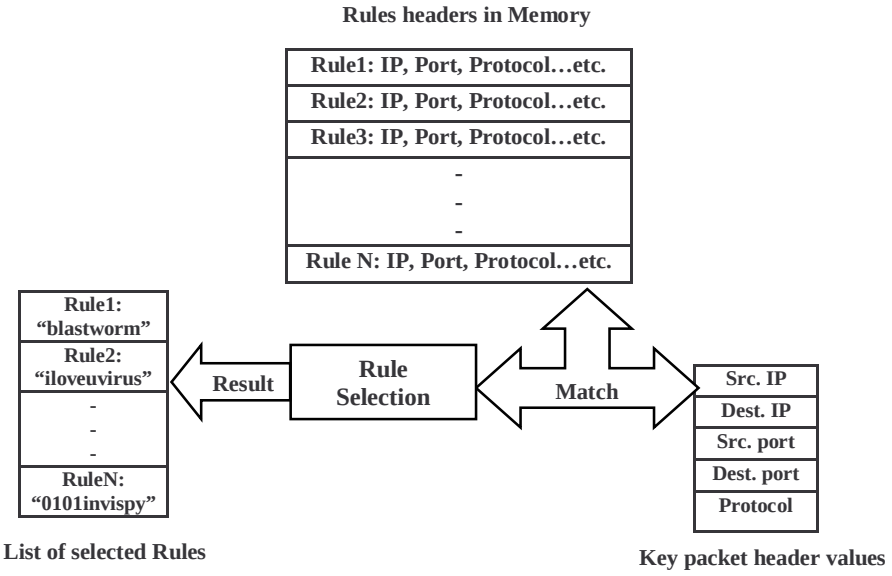


FIGURE 5.15: Rule selection

Next the rule evaluation is started on packet in packet header check and payload search component. Packet header check component evaluates only the header options (Non-payload options) of the subset of rules on packet header using SRT. On successful matching of header options it writes in SRAM the Rule ID of matching rule. Packet payload search component evaluates payload options of the subset of rules which involve searching attack patterns in packet payload using payload content modifier options. Before any payload option evaluation these content modifier options, any other options and packet payload are passed from MicroBlaze to packet payload search component via FSL bus.

The complete rule evaluation process on packet is now explained to get the greater understanding of operations performed by each component of hardware accelerator.

Packet Header check

Consider an example Snort rule in figure 5.7 for understanding its evaluation on packet payload with content: *abckill nowabdhgyppt*. First the payload options from this rule are extracted separately for passing on to PMHA. Only header option in this rule for packet header check is *ttl:<3;*. Packet header check component search the IP packet header *ttl* value. If the *ttl* value in a packet is less than 3 then the packet header check component writes the Snort *Rule ID* in SRAM and wait for the payload options evaluation result.

Packet Payload search

For Snort rule in figure 5.7, the only payload options to evaluate on packet payload is (*content:"kill now";*) which involves the search of attack pattern *kill now* in packet payload (abckill nowabdhgyppt). Content keyword modifier *depth* and *offset* of this rule indicates search for *kill now* only between byte-4 to byte-20 of packet payload. This search is carried by four modules of hardware accelerators with the support of Bloom filter pre-programmed with all attack patterns from Snort rules including the sort rule in figure 5.7 [4]. Figure 5.16 shows the payload search processing flow involving these four hardware modules and now explained in further detail.

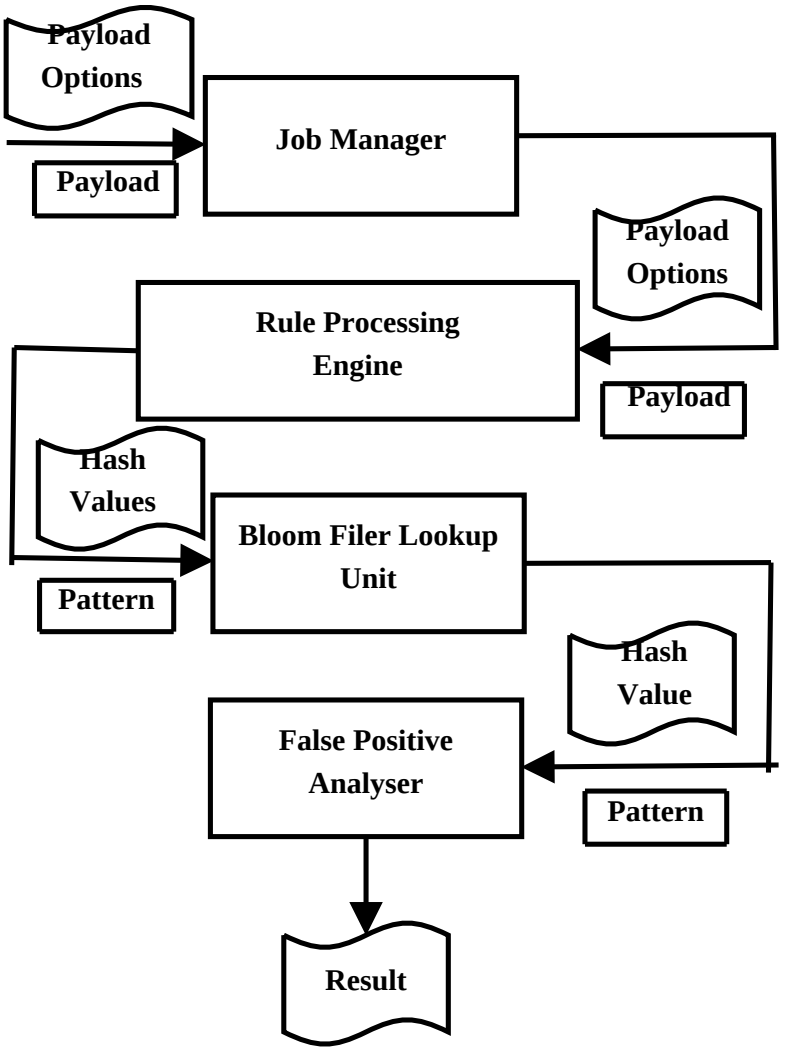


FIGURE 5.16: Hardware modules performing packet payload searching

Job Manager: Packet payload content and payload options are passed from MicroBlaze to Job manager module of PMHA. It is the first hardware that receives the payload

[4] A software program written in JAVA to extract attack patterns from Snort rules to create the hardware representation of Bloom filter

options and packet payload from MicroBlaze via FSL. Its main purpose is to manage the complete packet payload search process. It performs this by initiating packet payload search in the Rule Processing Engine and streaming the packet payload. Job manager not only initiates the packet payload search but also aid in pruning false positive match. All these functions are performed by Job Manager with the help of its five sub-modules: *Rule loader*, *Rule dispatcher*, *Match loader*, *Match dispatcher* and *Data feeder*.

Rule Processing Engine: This is where the actual payload search starts. Rule processing engine has sixteen *rule units* that are able to perform packet payload search for sixteen Snort rules in parallel. This is carried out with the help of its sub-modules: *hash module* which compute the hash values on substring of streaming packet payload for checking it's presence in Bloom filter.

Figure 5.17 shows the hash computation process on every 8-bytes substring of packet payload *abckill nowabdhgyppt* in rule unit for Bloom filter lookup.

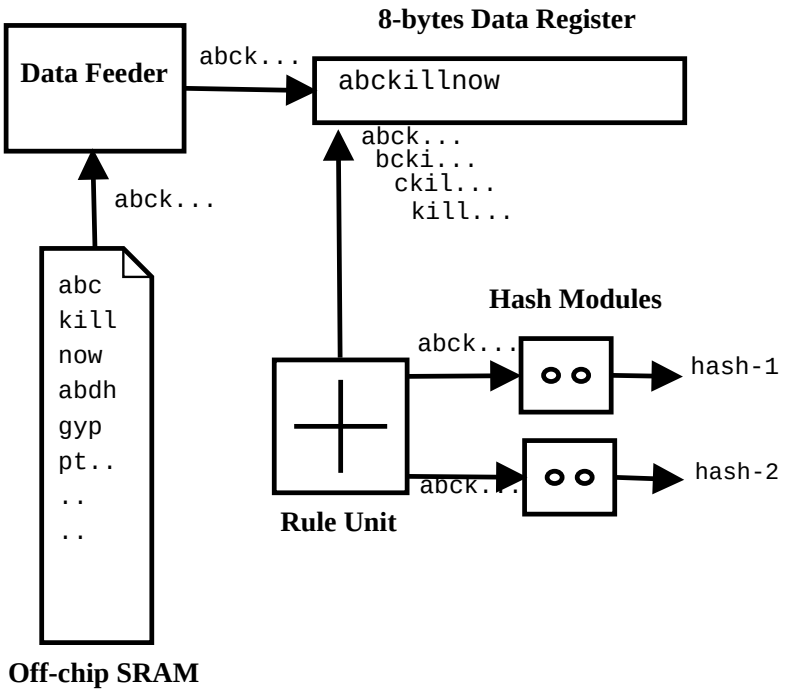


FIGURE 5.17: Rule units computing hash values

Packet payload bytes *abckill nowabdhgyppt* is streamed through 8-byte data register by Data feeder module which is synchronised with rule units. Rule unit then copies 8-bytes *(kill now)* of data register, signals the data feeder that data has copied. It then copied those 8-bytes to two hash modules operating in parallel. These Hash modules computes two hash values in just 2 clock cycles using hardware efficient XOR-based hash function. This hash value is passed on to the Bloom filter lookup unit for checking Bloom filter index for corresponding hash values. This process of hash computation is repeated until all 8-bytes substrings are processed up to packet payload byte-20.

Bloom Filter Lookup Unit: Assume that this module receives two hash values of *kill now* substring. Before any Bloom filter lookup this module actually computes further eight hash values using two hash values. This is carried out with the help of *2-to-N hash module* in just 2 clock cycles. Figure 5.18 shows the hash computation process.

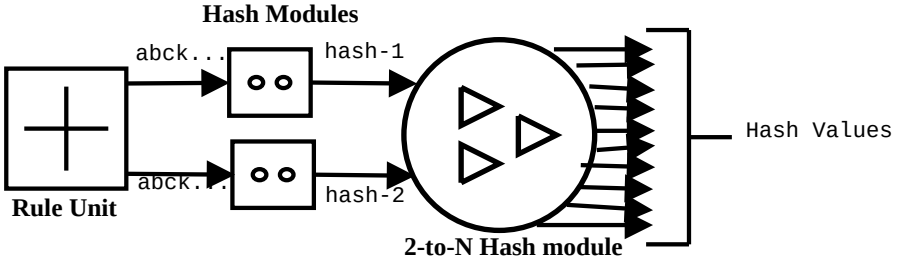


FIGURE 5.18: 2-to-N hash module computing ten hash values using two hash values

All hash values are now copied to Bloom filter probe unit for checking Bloom filter index for corresponding hash values. The Bloom filter is constructed in dual port FPGA Block RAM. The boom filter lookup process for 10 hash values takes further 5 clock cycles as shown in figure 5.19.

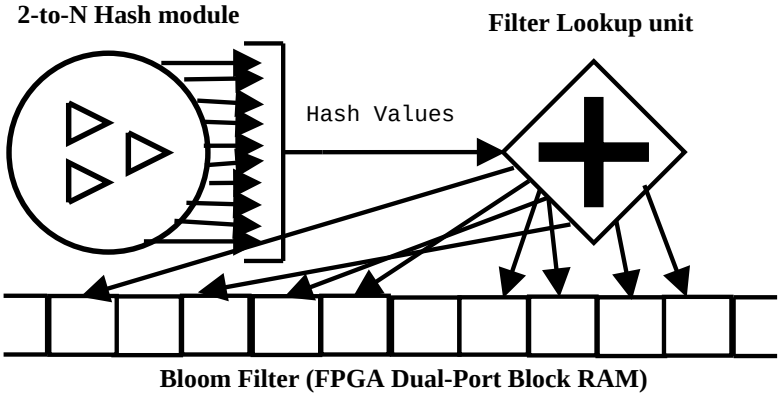


FIGURE 5.19: Bloom filter index checking with corresponding hash values

For all ten *kill now* hash values, Bloom filter index corresponding to hash value is found to be 1 because the pattern *kill now* in Snort rule 5.7 is parsed and programmed in Bloom filter. This match however may be a false positive due to nature of Bloom filter. A hash value of *kill now* is then copied to False Positive Analyser Module to verify the match.

False Positive Analyser Module: This module prune the false positive match. It performs this pruning with the help it's two sub-modules: *Hash table lookup unit* and *comparator circuit* and hash table in SDRAM where all patterns and associated Snort Rule IDs are stored. To check *kill now* is true match, the hash table lookup unit uses the hash value of *kill now* to fetch content of hash table index for corresponding hash value. It then passed the content to the comparator. If comparator found any Rule ID then it start comparing payload substring *kill now* with a pattern from hash table. The

pattern comparison process is a simple brute-force pattern matching that performs four bytes comparison per clock cycles. If all *kill now* bytes matches the pattern then it is declared as a match. The Snort Rule ID of this pattern is then written to SRAM for processing of detection result on MicroBlaze. Figure 5.20 shows this process.

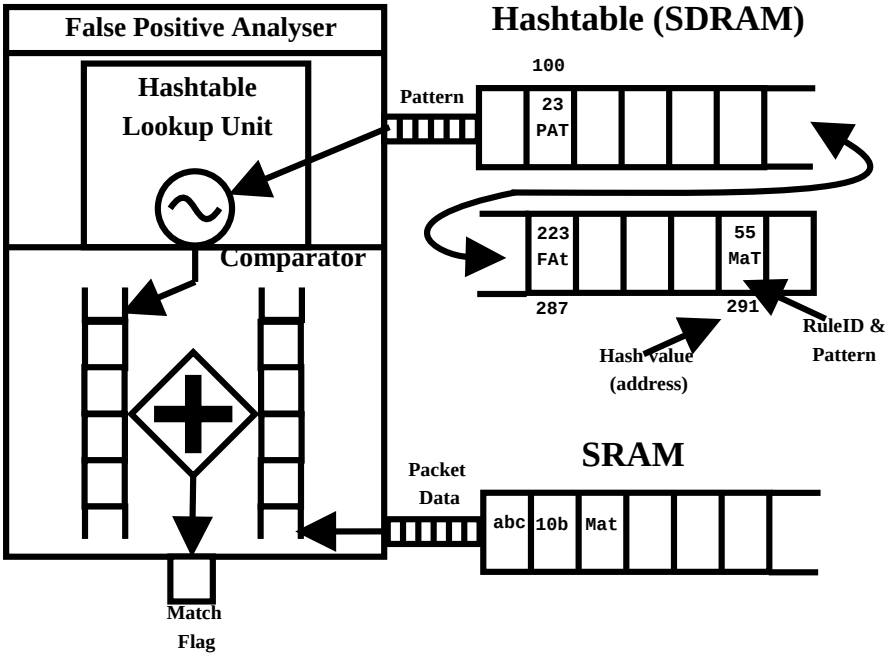


FIGURE 5.20: False positive analyser with hash table lookup unit and comparator circuit

Snort Rule Match

Packet header check and packet payload search write the Rule IDs of matching header options and payload options of Snort rules. Only those Snort rule is declared a match when the same Rule ID of rule is found in packet header check result as well as in packet payload search result.

5.3.3 Implementation

Packet payload search or PMHA are attached to high speed FSL bus. The communication between hardware accelerators on FSL bus and rest of the Snort detect engine component on MicroBlaze is facilitated with the help of HandelC library (fsl_t1.hcl) which contains the source code for controlling FSL bus and enabling communication between hardware accelerators and software on MicroBlaze processor.

Pattern Matching Hardware Accelerator (PMHA)

HandelC PMHA is implemented using the necessary data structure and macro procedure (Section 4.3.4). Its data structure comprises of only one 1-bit status registers (busy register). A busy status register initially set to clear or zero to indicate that no Snort rules are needed processing in hardware accelerator. Figure 5.21 shows the architecture of PMHA.

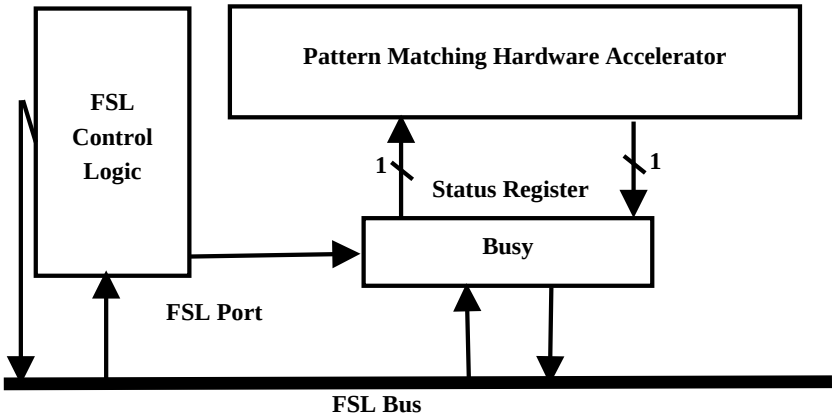


FIGURE 5.21: Pattern Matching Hardware Accelerator (PMHA) architecture

The communication between packet capture hardware accelerator and the Packet Sniffer part on MicroBlaze is synchronised and supported by 1-bit status registers, methods and macro procedures containing hardware accelerator logic. Snort modified Detection Engine component on MicroBlaze first calls the *get_fsl_nbread(int fslport,int bitvalue)* read method with FSL port number and variable to read in the content of register. This invoked the callback read macro procedure *FslSlaveRead()* hardware accelerator that reads the content of status register and writes it's contents to the FSL data bus. If the check on variable found the register contents zero then Detection Engine component starts sending the payload options of rule and packet payload content towards the accelerator hardware. The hardware accelerator then evaluates the payload options on packet payload and once finished write the detection result in SRAM BANK-0 and clears the status register content by writing 0.

One of the important parts of Bloom filter based pattern matching hardware implementation is the hash function. Oncoming section contains the detail discussion on hash function implementation and explanation on computing large number of hash values readily and efficiently in hardware.

Optimal hash function

One of the main considerations of the Bloom filter based pattern matching implementation in hardware/FPGA is the selection of appropriate hash function. Ideal hash function should consume minimal FPGA power and resource as well as produced lower or atleast theoretical false positive rate (Section 6.4). To achieve this effectively a class of universal hash function called H_3 that exploits bit-wise logical operations [116, 117] is selected for implementation. Such a hash function is efficient in terms of hardware resource consumption and due to simple bit-wise logical operation it can be readily and efficiently implemented in FPGA. Dharmapurikar et al. [81] also implemented this hash function for their Bloom filter based pattern matching hardware implementation on FPGA [81]. Figure 5.22 illustrates the equivalent hash function circuit implementation of HandelC code.

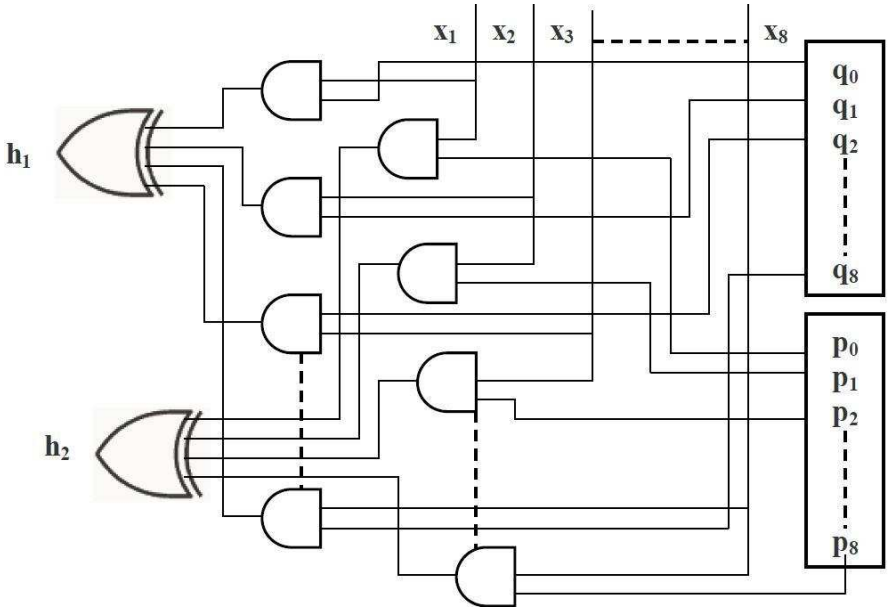


FIGURE 5.22: Hash calculator circuit design

This shows that for any input bit-pattern X with i number of bits represented as,

$$X_i = [x_1, x_2, x_3, ...x_8] \tag{5.3}$$

The k^{th} hash value H_k for X_i is calculated as,

$$H_k = [(q_0 \cdot x_1) \oplus (q_1 \cdot x_2) \oplus, (q_2 \cdot x_3) \oplus ... \oplus (q_7 \cdot x_8)] \tag{5.4}$$

Where $\cdot$ denotes the binary AND operation and $\oplus$ the exclusive OR operation. $q_{(0..j-1)}$ is a vector of random integer values in the range of *1* to m. The vector values are generated

by a software program that developed in JAVA programming language. The vector values then copied from software program into the index of HandelC array (registers). Similarly, another k^{th} hash value H_k for bit-pattern X_i can be calculated with another vector $p_{(0..j-1)}$ of random integer values in the range of *1* to m as,

$$H_k = [(p_0 \cdot x_1) \oplus (p_1 \cdot x_2) \oplus, (p_2 \cdot x_3) \oplus ... \oplus (p_7 \cdot x_8)] \tag{5.5}$$

This vector $p_{(0..j-1)}$ of random integer values is also generated by the same software program and copied into the index of another HandelC array (registers).

Two Hash Function Solution

Equation 5.2 suggested that the ratio of the number of bit-pattern n to program in a Bloom filter to the total number of bits m in Bloom filter is the number of bits allocated to each bit-pattern in a Bloom filter. Taking *ln2* of this value decides the optimal number of k hash functions needed to compute for each bit-pattern to insert/query them into Bloom filter. Let suppose if there are n = 2385 number of bit-patterns and the number of bits in Bloom filter is m = 32768, then on average $\approx$ 14-bits per keyword is allocated in Bloom filter (m/n = 13.73 $\approx$ 14). For this ratio (m/n) value, the number of hash functions needed to compute per bit-pattern for insert/query is k = 13.73 $\times$ ln2 $\approx$ 10 (where, ln2 = 0.69314). This number of hash function is difficult to implement in FPGA and for query intensive application like SB-NIDS pattern matching it may become bottleneck. Simple solution to this problem is to reduce the number of hash functions by reducing the number of bits allocation per keyword in Bloom filter. However, this simple solution would cause increase in false positive probability that also result in comparing large number of bit-patterns with brute-force (bit-by-bit) pattern matching. For example: false positive probability according to equation 5.1 is, 1-e$^{-\frac{2385\times10}{32768}}$ $\approx$ 0.001365, if lower value of k is chosen by reducing the number of bits allocation per bit-pattern in Bloom filter to m/n 6.85 $\approx$ 7 by lowering the number of bits in Bloom filter to m = 16384, then value of k would be, k = 7 $\times$ ln2 $\approx$ 5 (where ln2 = 0.69314). This will increase the false positive probability of Bloom filter to 1-e$^{-\frac{2385\times7}{16384}}$ $\approx$ 0.03695.

To compute such a large number of hash values, a very different technique is used that can yield an effective speed up of pattern matching in hardware for SB-NIDS. It is the first time this technique is use for SB-NIDS pattern matching in hardware which is originally proposed by Kirsch [6] for Bloom filter and related data structures. According to this technique, only two hash functions are necessary to effectively implement a Bloom

filter without any increase in asymptotic false positive probability. With this idea two hash functions $h_1(x)$ and $h_2(x)$ can simulate more than two hash functions of the form,

$$g_i(x) = (h_1(x) + ih_2(x)) \bmod m \tag{5.6}$$

i ranges from *0* to *k-1* for k number of hash values. For example: if i = 9, then this will produce ten hash values. Every hash value is taken modulo of the Bloom filter size (m number of bits) to ensure the values within the range of m.

In summary, it is the first time ever a PMHA for SB-NIDS implemented with Snort application specific knowledge and with efficient hash function computation technique. Also unlike other Snort Rule Processing System (Section 3.5.3), this PMHA is integrated with the Snort SB-NIDS prototype on hybrid HandelC-MicroBlaze embedded processing platform to perform a complete rule processing. It is also this hardware accelerator that is implemented with the largest number of patterns (7176) from Snort rules and compactly stores them in FPGA block RAM.

5.4 Final Optimisation of Snort Port (MMU-Snort III)

This section contains explanation of further improvements to the hardware accelerators which resulted in MMU-Snort III prototype. This involved hardware accelerator architecture and algorithm improvement:

- An algorithmic and architectural improvement of PMHA algorithm to optimise pattern pruning process.
- An extension to existing PMHA algorithm to efficiently search longer size patterns ($>$ 64 bytes).

A brief analysis is presented that shows the issue with the PMHA algorithm. This analysis also covers how the longer patterns can be efficiently search with PMHA algorithm.

5.4.1 Analysis

This analysis reviewed the need of algorithmic changes of existing PMHA algorithm. This will result in increasing the speed of PMHA algorithm and improved longer patterns search in packet.

Rule Evaluation revisit

The modified Snort Detection Engine on hybrid HandelC-MicroBlaze processing platform performs rule evaluation which is summarised as follows:

- Snort rule selection by matching key packet header values (IP, Port, and Protocol etc.) with all rule headers on MicroBlaze.
- Selected rule evaluations, this involve rule header options evaluation on packet header on MicroBlaze and rule payload options evaluation on packet payload on FPGA.

The header options evaluation is trivial which involve checking numeric values in packet header. In case all header options match, the Snort Rule ID is recorded. The payload options evaluation which is the packet payload search of attack pattern is complicated and computationally demanding process. It involved hash value computation of substring from packet payload and Bloom filter index lookup for corresponding hash value. If substring found to be programmed in Bloom filter and also matches with a pattern from hash table, its Snort Rule ID is recorded. The header options and payload options result is then processed on MicroBlaze.

Algorithm

Section 5.3.2 contains the detail explanation of payload options evaluation on packet payload using four of the PMHA modules. It is observed that payload option evaluation in False Positive Analyser module performing lot of unnecessary substring comparison with patterns stored in hash table. This is highly susceptible to DoS attacks because if attacker sends a carefully crafted packet with patterns that repeatedly triggers the brute-force searches. This can become further worst with longer length patterns. The main reason of excessive number of pattern comparison is that there is no mechanism exists to further prune the result before full pattern comparison. Figure 5.23 shows this issue.

Only Rule IDs found on both sides of the rule evaluation (header options and payload options) are the only rule match, so some of the payload search is resulting in waste of clock cycle or time.

Example: Following example further clears this problem. Consider two Snort rules (Figure 5.24) which is parsed by Detection Engine to construct the SRT like the one in

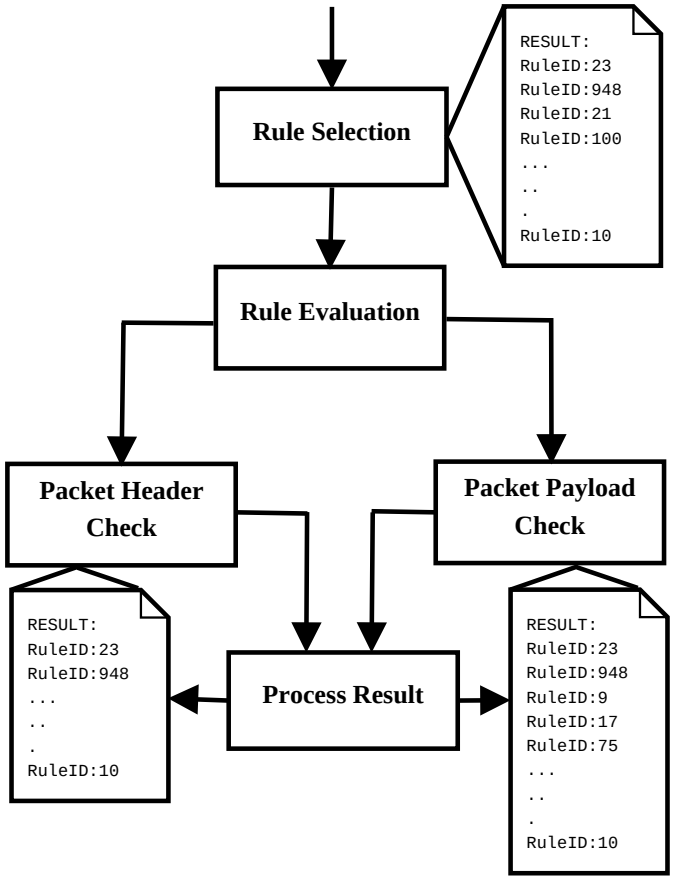


FIGURE 5.23: Rule selection and evaluation result

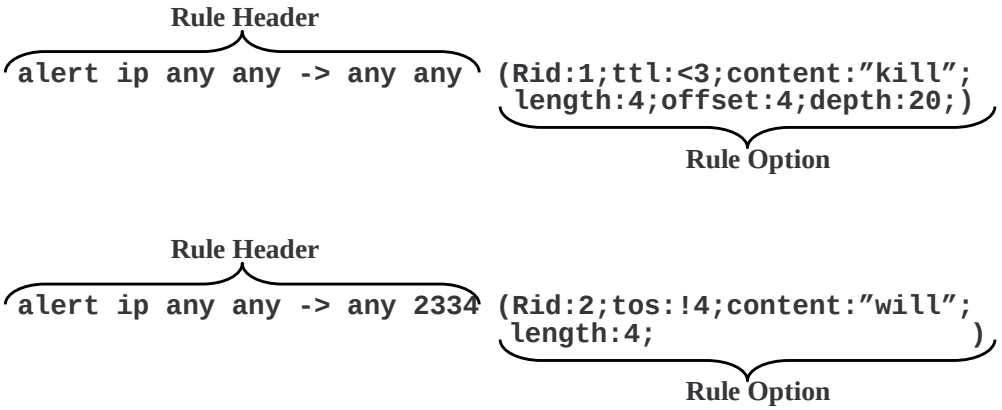


FIGURE 5.24: Example Snort rules

figure 5.5, the patterns are programmed in Bloom filter and the Rule ID (Rid) along with pattern are added to hash table.

Imagine only the first rule (RuleID:1) header match with key packet header values and selected for evaluation on packet. Rule header options (ttl:<3) evaluation for this rule using SRT also found the packet IP header has *ttl* values <3. This means that this rule header options is declared a match and it's RuleID:1 is recorded. Now the remaining part of rule evaluation which is a packet payload search or payload options evaluation carried

out in PMHA using Bloom filter. It has to search all four bytes substring in packet data from byte 4 to byte 20. If the packet payload is "abcdkillsushd688willfhfjhk10" then packet payload search will find both "kill" and "will" substring programmed in a Bloom filter. For false positive check, it will also find the Rule ID and pattern in hash table which will trigger the brute force pattern matching for both substrings with pattern. The brute-force matching involves fetching the patterns from hash table in SDRAM and comparing it with substring found in Bloom filter. The packet payload search result in the end declared both patterns a match and mentioned the associated Rule IDs. Although both substring are found in Bloom filter and also match with pattern using brute-force matching, but the reality is it is only "kill" in the end that is match. The reason is simple, as 'kill" belongs to the first rule and it is only first rule that was selected for evaluation as well as first rule header option check on a packet also result in a match. The second rule (RuleID:2) was not selected at all for evaluation on a packet and so "will" is not a match.

In summary, algorithm successfully able to perform rule evaluation but has issues which resulting in lot of unnecessary brute-force matching. A design changes in pattern matching algorithm was suggested to remove this issue and boost up algorithm performance (Section 5.4.2). Another issue of the pattern matching algorithm is the poor performance with longer pattern (> 64 bytes). An analysis is now present which explains how to efficiently search longer length patterns with PMHA.

Snort rules

First an analysis on Snort rules is presented here to find out the number of over 64 bytes pattern that the PMHA does not search efficiently.

There are total 7876 unique patterns in Snort rules released in June 2008. Out of 7876 patterns, 7170 patterns are 64 bytes and less which hardware accelerator can search efficiently using Bloom filter. The rest of the 706 patterns are over 64 bytes which hardware accelerator does not search efficiently. A clear spike in the graph in figure 5.25 at x-axis (180) value can be notice which indicates that there are 439 (y-axis) patterns of 180 bytes (x-axis) in length.

Now the total number of over 64 bytes pattern is identified and solution is needed to search them efficiently.

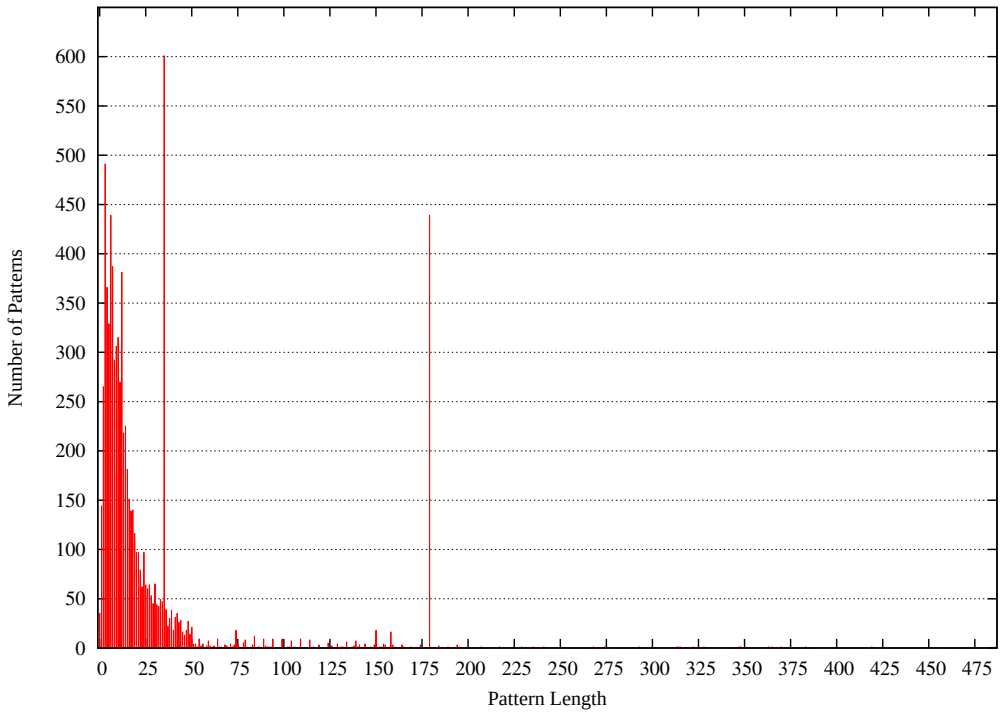


FIGURE 5.25: Patterns from Snort rules with their length

Longer patterns (> 64 bytes)

A software program was written in JAVA using JDK 1.6.0 to analyse the longer patterns in Snort rule. This analysis concentrated on finding out how many more patterns would result if patterns over 64 bytes are broken down in 64 bytes chunks. Figure 5.26 shows the graph that demonstrate this.

This graph clearly shows the effective increase in number of pattern as a result of breakup. Table 5.2 shows the exact summary of the total number of increase in patterns as a result of over 64 bytes pattern breakup.

TABLE 5.2: Total number of patterns

Summary	
Total Pattern	7876
> 64 bytes (Before breakup)	706
> 64 bytes (After breakup)	1980
New Total	**9150**

The breakup of longer patterns into 64 bytes chunks would be used to optimise the search of longer pattern. This will require only minor changes in PMHA algorithm. Before this is explained, pruning optimisation in False positive analyser module will be explained which required minor algorithmic and architectural changes of PMHA.

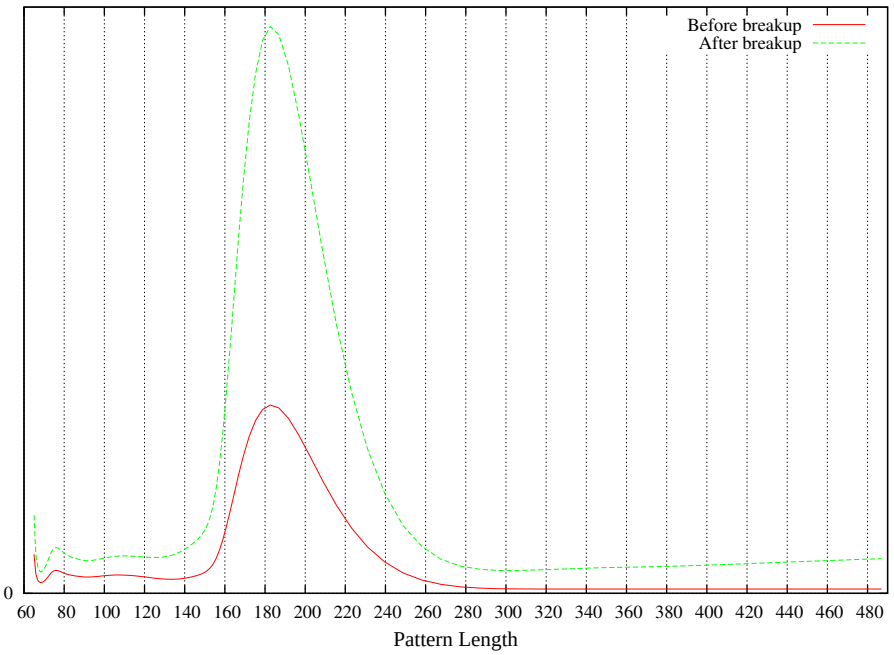


FIGURE 5.26: A line graph showing the increase of patterns after breakup

5.4.2 Design

This design changes is carried out in PMHA (Section 5.3.2). These design changes involved minor algorithmic and architectural changes mainly concerned improving the performance of PMHA.

Algorithm Modification

In section 5.4.1, the problem with the pruning process in False positive analyser hardware module of PMHA is illustrated. This highlights the need to come up with some kind of fix in false positive analysis hardware module that can reduce the unnecessary number of pattern comparison with brute-force searching. The algorithm fix is now explained.

Figure 5.27 shows flowchart of old and new pattern search algorithm.

The changes in the new algorithm can be notice in the fourth processing stage. A new check is inserted at this stage to prune further the substring match before actual brute-force comparison. This new check make sure that only those substring are compare with patterns in hash table using brute force comparator whose RuleID from hash table matched with the RuleID of the rules selected for evaluation in Rule Selection component (Figure 5.15). The modified algorithm will now result only with little number of RuleIDs as shown in figure 5.28.

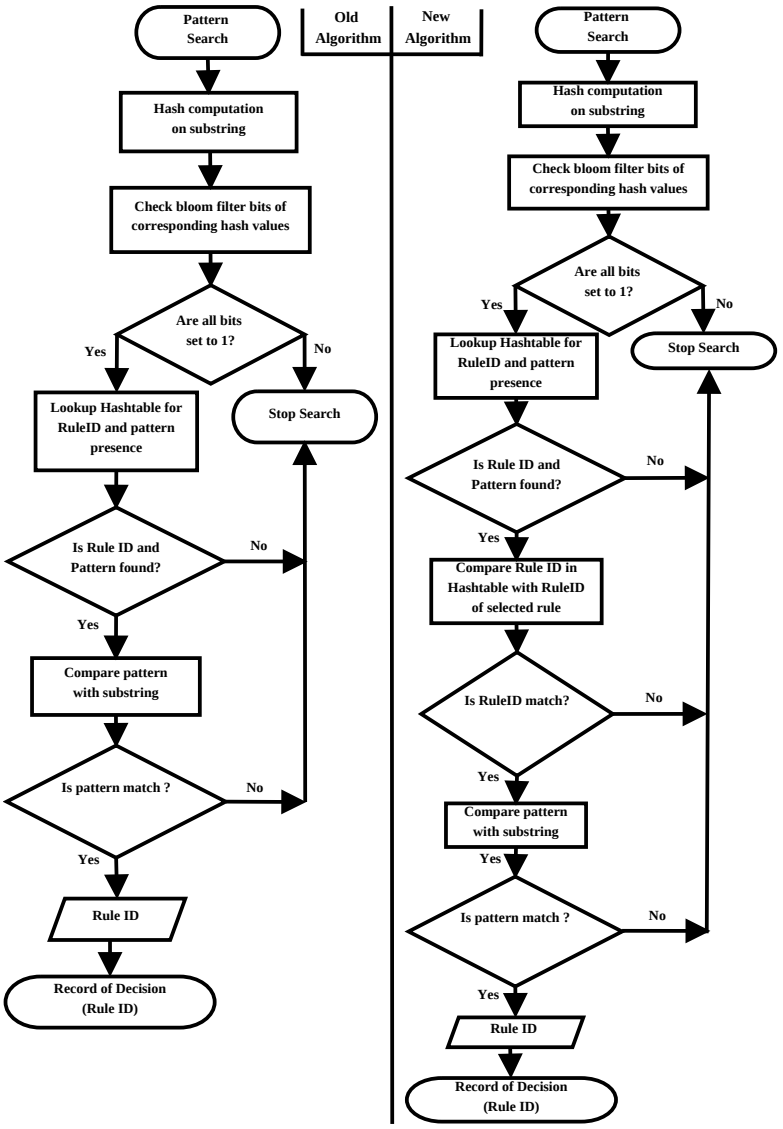


FIGURE 5.27: Old and new Pattern Matching algorithms on FPGA

The new modified algorithm can now be made clear further with the help of following example.

Example: Consider the Snort rule in figure 5.24. Suppose the second Snort rule (RuleID:2) header matched with key packet header values and so selected to carry out evaluation on packet. Let suppose that header options (tos:!4) check in IP packet header also found the *tos* value other than 4. This means the second rule is declared a match for header options evaluation and its RuleID (RuleID:2) is recorded. Now the remaining part of rule evaluation which is a pattern search or payload options evaluation is carried out in FPGA using Bloom filter. Suppose the packet payload is "abcdkillsushd688willfhfjhk10" then packet payload search will find both "kill" and "will" substring programmed in a Bloom filter. On finding the first substring "kill" in Bloom filter an additional check is

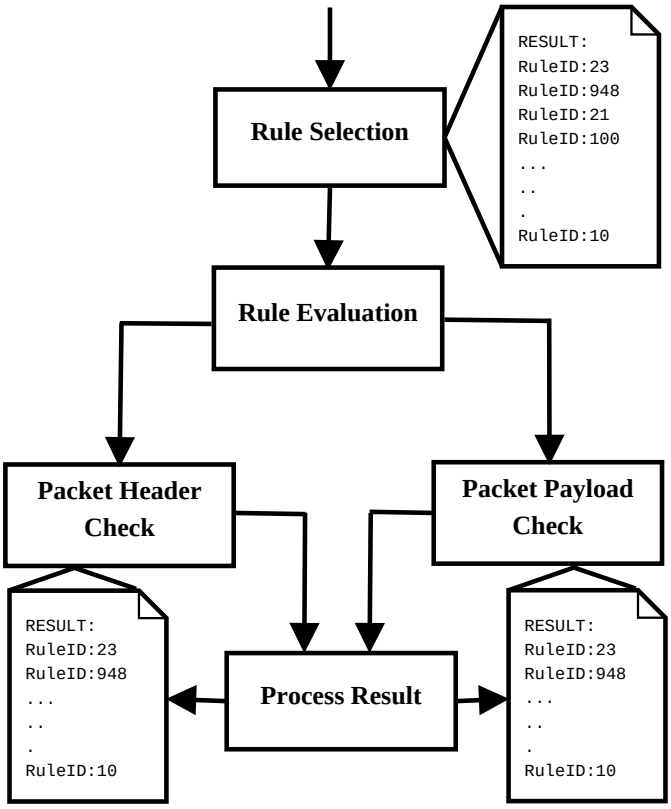


FIGURE 5.28: Rule selection and evaluation result with modified algorithm

carried out now in this stage to confirm if the patterns found in Bloom filter that has associated RuleID in hash table which matches with RuleID of selected rule for evaluation. This comparison result will not match as the RuleID associated with "kill" is RuleID:1 and the search is carrying out for second rule RuleID:2. This means a straightforward single clock cycle check has removed the need to compare substring with the pattern from hash table using brute-force matching with comparator. For pattern "will" the RuleID comparison would result a match. This time the pattern is fetched from hash table in SDRAM and the brute force pattern matching of substring and pattern is performed. The comparison in this case would result in a match and Snort RuleID:2 is recorded. As both header options and payload options evaluation result output the same RuleID:2 then this rule is declared a match.

Architecture Modification

An extra pruning step is implemented in the False positive analyser hardware module. The false positive analyser module is comprises of hash table lookup unit and a comparator unit (Figure 5.20). The hash table lookup unit is the module in which the actual architectural changes are made. This is carried out with a simple HandelC code snippet which compares the RuleID from hash table with the RuleID of the rule selected

in the rule selection component. Figure 5.29 shows this architectural modification in which hash table lookup unit now has a *RuleID comparator unit* to carry the RuleIDs comparison which takes only 1 clock cycle.

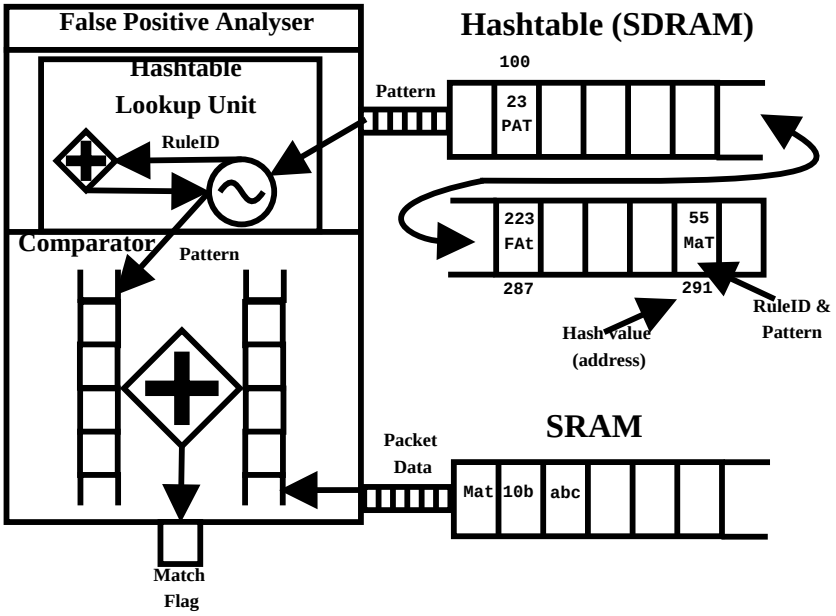


FIGURE 5.29: Modified false positive analyser with Hash table lookup unit and Comparator circuit

Long Pattern

To optimise the pattern matching of total 706 longer patterns (> 64 bytes) a simple solution is proposed instead of different algorithm. Any new algorithm to deal separately with longer pattern would make system more complicated as well as it's implementation would require more FPGA resources (area/space) on current development board which may exhaust all FPGA resources and does not synthesize the full design on FPGA. Therefore, an extension to the current PMHA (Section 5.3) seems appropriate solution to deal efficiently with larger pattern. This is achieved by breaking the patterns into smaller chunks and searches them using the same pattern matching algorithm accordingly. This would require only minor changes in PMHA algorithm which now explained.

Algorithm

The breakup of pattern will not only increase the number of patterns but also slightly increases the Bloom filter false positive rate. However, the pattern breakup would result in increase of the overall throughput of the PMHA so considered and accepted to extend the pattern search algorithm to search differently the longer patterns.

PMHA remains unmodified. The main changes were carried out in pattern search algorithm and hash table structure for dealing with longer pattern. Figure 5.30 shows the flowchart of the algorithm that deals efficiently with longer patterns.

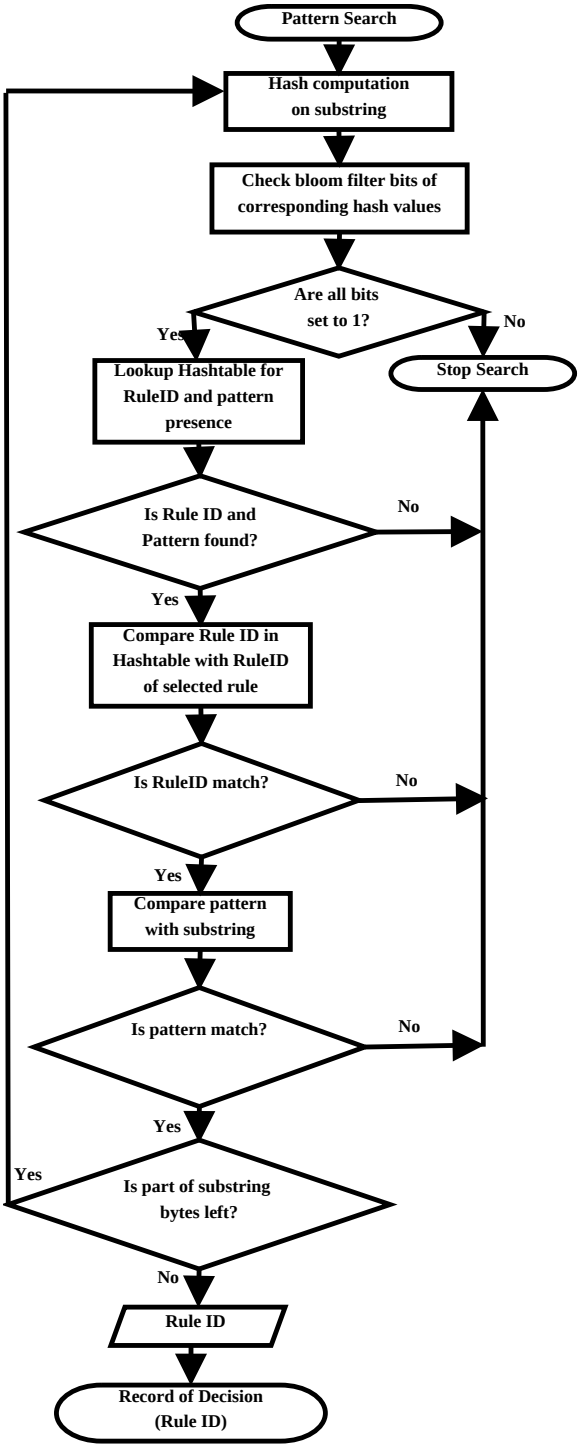


FIGURE 5.30: Pattern matching algorithm flowchart for longer (> 64 bytes) pattern

The pattern matching algorithm is slightly modified to deal with longer pattern search in a packet differently. The modified algorithm makes sure to compute the hash value of up to 64 bytes pattern. Pattern search longer than 64 bytes now search in chunks.

An additional check is added to algorithm before declaring a substring search result. This check makes sure that all chunks are processed before deciding the match result. If during search any chunk does not found programmed in Bloom filter or declared false positive in false positive analyser hardware module then the further chunks or remaining bytes of substring is not search anymore.

The modified algorithm searches the longer pattern more quickly and also consumes lesser number of clock cycles. This modification and pruning helped to increase the overall throughput of pattern search in PMHA (Section 6.4)

5.5 Chapter Summary

This chapter began with the description of novel SB-NIDS prototype architecture or MMU-Snort I which is designed and implemented by porting Snort (ver. 2.6.1.4) SB-NIDS software package on hybrid HandelC-MicroBlaze based embedded processing platform. This was followed by the description of the PMHA which is the most computationally intensive operation of SB-NIDS. This novel PMHA design is based on Bloom filter search approach and first time ever implemented with Snort application specific knowledge and with efficient hash function computation technique. Unlike other Snort Rule Processing Systems (Section 3.5.3), this PMHA is integrated with the Snort SB-NIDS prototype on hybrid HandelC-MicroBlaze embedded processing platform to perform a complete rule processing which result in MMU-SnortII prototype. This hardware accelerator has been implemented with the largest number of patterns (7876) from Snort rules and compactly stores them in FPGA block RAM (Section 3.5.3). Finally, a further algorithmic and architectural improvement of PMHA is presented to improve pattern pruning process and to add support to efficiently search of longer patterns (> 64 bytes).

Chapter 6

Results and Analysis

The basic goal of any software system testing and evaluation is to ensure that the system works as per the functional requirements as well as meeting other requirements (Section 1.3). Therefore, SB-NIDS prototype and its hardware accelerators are evaluated to identify any improvements and issues in processing. The test results are also compared with the state of the art systems to verify the improvements of the proposed design.

First the functional test is performed to verify error free execution of MMU-Snort I or modified Snort port on hybrid HandelC-MicroBlaze embedded processing platform (Section 5.2). Then the performance test is performed which involved comparing MicroBlaze CPU cycles executing Snort port with general purpose processor CPU cycles executing the same version of Snort.

Performance test of MMU-Snort II or Pattern Matching Hardware Accelerator (PMHA) (Section 5.3) and MMU-Snort III (Section 6.4) is performed to determine the improvement. This is carried out by obtaining the attack patterns memory space amount, determining the effect of different hash functions and packet analysis throughput. The memory space result and throughput results are compared with the state of the art pattern matching systems (Section 6.4.2).

6.1 Chapter Roadmap

The rest of the chapter is outlined as follows:

- In section 6.2, the experimental testbed network topology is explained which is used for the testing of the prototype SB-NIDS and hardware accelerators.

- In section 6.3, the SB-NIDS prototype or MMU-Snort I testing and evaluation results are presented. The prototype solution is tested for functional and performance test. The functional test is carried out to determine the correction functionality. The performance test is carried out to identify the performance improvement. This is carried out by counting CPU clock cycles. In the end the FPGA synthesis results summary is presented.

- In final section 6.4, PMHA or MMU-Snort II and MMU-Snort III performance testing and evaluation results are presented. The performance test involved memory requirement test, throughput test, false positive rate test and FPGA design/space test. Finally, the throughput and memory test is compared with state of the art pattern matching solutions.

6.2 Experimental Testbed

The experimental testbed was prepared to carry out the system testing. This test network consists of two PCs and RC300 board connected together as shown in figure 6.1.

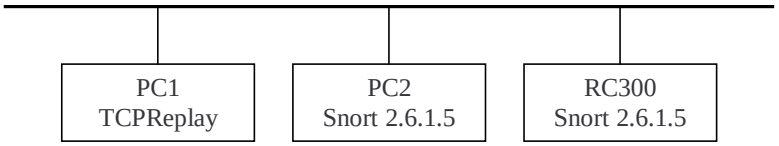


FIGURE 6.1: Topology of experimental test network

PC1 with Debian Linux 2.6.18 was installed with the TCPreplay (version 3.3.2) traffic generator software. TCPreplay reads logged packet from a network trace file *(tcpdump format)* and sends the packet through an Ethernet interface. PC2 with Ubuntu Linux 8.04 (2.6.25) was installed with Snort version 2.6.1.5 and the Performance Application Programming Interface (PAPI) [118] in order to obtain an accurate processor execution cycle count. The prototype system based on Snort version 2.6.1.5 implemented on an RC300 board was also connected to the test bed. On MicroBlaze, an execution cycles count is obtained using method *(XTmrCtr_GetValue (TmrInstance, TIMER_ID))* defined in *xtmrctr.h* header files. The test network is a 100MB subnet that is private and isolated from all other networks. The system is tested throughput against network trace files from MIT Lincoln Lab's 1998 DARPA offline Intrusion Detection evaluation and from shmoo group capture the flag project [107, 119].

6.3 Testing and Evaluation of Snort Port (MMU-Snort I)

First the MMU-Snort I functional test is performed and then the performance test. The testing environment for both test is the same as shown in figure 6.1.

6.3.1 Functional Test

The functional testing was performed in two phases with two different configurations. In the first phase, both systems were configured with the same Preprocessor components (frag2, Stream4, Telnet, DNS and sfPortscan) and number of rules. They were then tested twice against two different network trace files from MIT Lincoln Lab's 1998 DARPA offline Intrusion Detection evaluation. In the second phase, again both systems were tested twice with the same network trace file but each time configured with different parameters. The first time both were configured for 5 Preprocessor components (frag2, Stream4, DNS, Telnet, and HTTP Inspect) and 5320 rules. On the second time both systems were configured for 6 Preprocessor components (frag2, Stream4, sfPortscan, FTP, Telnet, DNS) and 4747 different rules.

For both phases of experiments the detection result summary (Protocols breakdown) and alert messages generated by both systems are compared. It was observed that both NIDS produced the same alert messages *(Rule Security ID (SID))*. The detection summary also demonstrates that the types of packets analysed and number of alerts generated were identical. Thus, the prototype passes the operational test.

6.3.2 Performance Test

Performance test helped to determine exactly the computationally significant part of Snort source code as well as help in determining the packet analysis speed improvements. This is actually determined by CPU cycles count test.

CPU Cycles count

MicroBlaze cycle count executing Snort port on RC300 board and PC with general purpose processor executing original Snort package is obtained. Both systems were configured with the same parameters and number of rules.

The system was tested against three different network trace files. Two of these files are from MIT Lincoln Lab's 1998 DARPA offline Intrusion Detection evaluation and one

from shmoo group capture the flag project. Results obtained on both systems for all three data files are shown in figure 6.2.

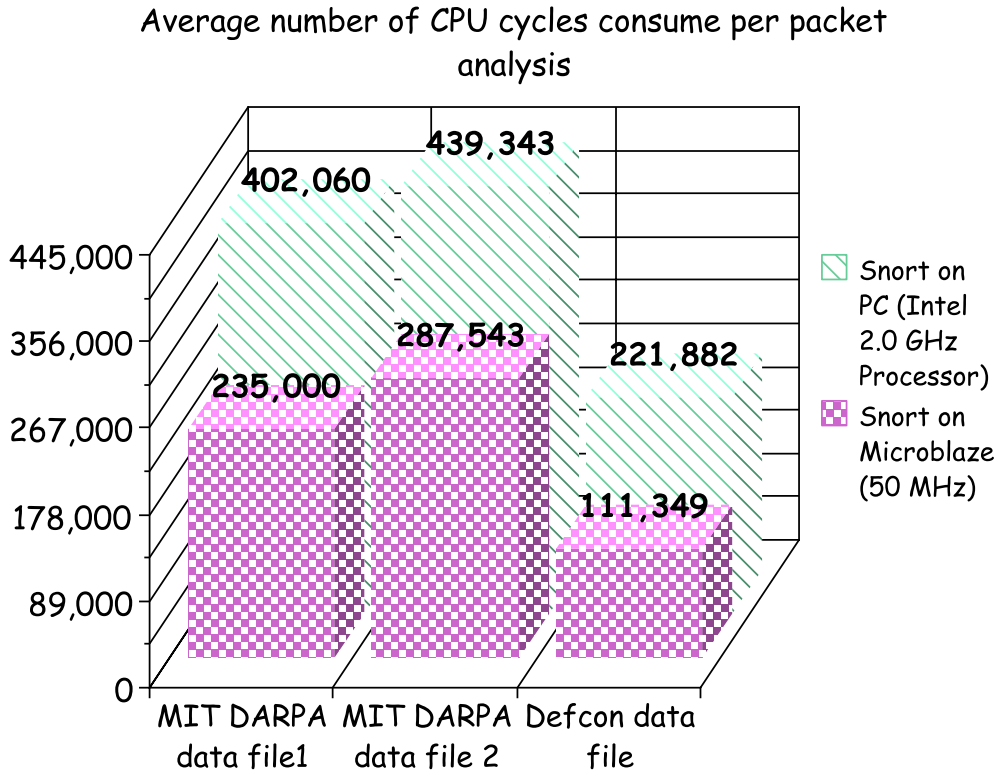


FIGURE 6.2: Snort CPU cycles comparison

The average CPU cycle count on the PC per packet is much higher than prototyped Snort system on RC300 board. This is mainly because of differences in how packets are captured due to the close coupling of the network interface on the FPGA board to the MicroBlaze core and also due to some restructuring in the design. Another main reason of lower clock cycle in Snort port on RC300 is the differences of prototype system architecture. For example, MicroBlaze method *OPBREADUINT32()* to read 32 bit data from SRAM require only 16 clock cycles over OPB Bus and MicroBlaze method *OPBWRITEUINT32()* to write 32 bit data to SRAM requires only 19 clock cycles over OPB Bus.

Lost CPU cycles

The above test presents the average CPU cycles consumed for packet analysis on both architectures with three different data files. In order to identify the Snort code sections that consume computationally significant portions of CPU cycles methods related to packet capture, pattern matching, packet classification and packet decoding on both architectures are instrumented. This experiment is conducted only for DARPA data file

1 6.2. Figure 6.3 and 6.4 shows the average number of CPU cycles required per packet for these main methods that usedSnortt for packet analysis.

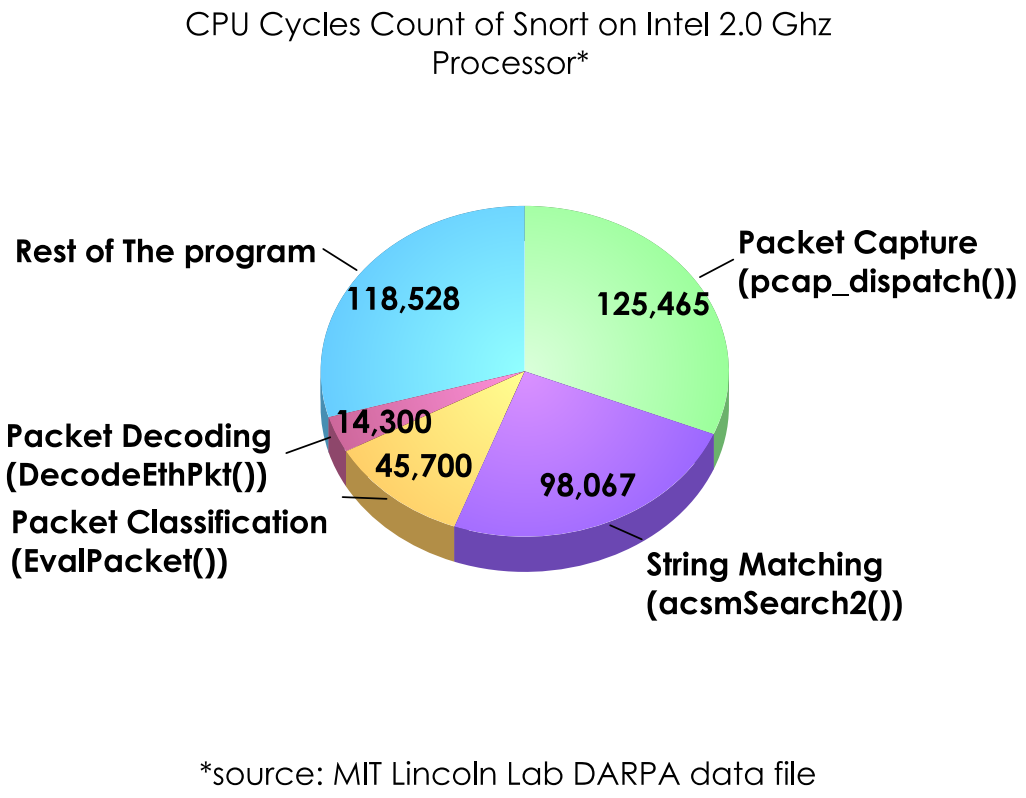


FIGURE 6.3: CPU cycles count of Snort on PC

In these results the only major difference in the number of CPU cycles was noticed for packet capture method. Packet capturing on PC requires copying of packet from kernel memory to a user-level application buffer that normally consumes large number of CPU cycles due to buffering and user-kernel protection mode switching. In contrast to the PC, the packet capture method of prototype system requires only 1 clock cycle to capture a packet byte from the Gigabit Ethernet interface and another cycle to store that byte in SRAM. The number of CPU cycles required accessing SRAM from MicroBlaze over the OPB Bus and to copy that byte to Snort application buffer on average requires accurately 16 clock cycles. This may vary depending on the number of peripherals on OPB bus.

It also observed that other methods on both architectures do not show any major difference in computational requirement in terms of the number of CPU cycles. However, it is clear from results that the number of CPU cycles consumed for the same methods on MicroBlaze is slightly lesser in number than that of Snort on PC. It is highly likely that this slight different is due to differences in their hardware architectures as MicroBlaze soft processor core on RC300 board is executing on tightly coupled environment where

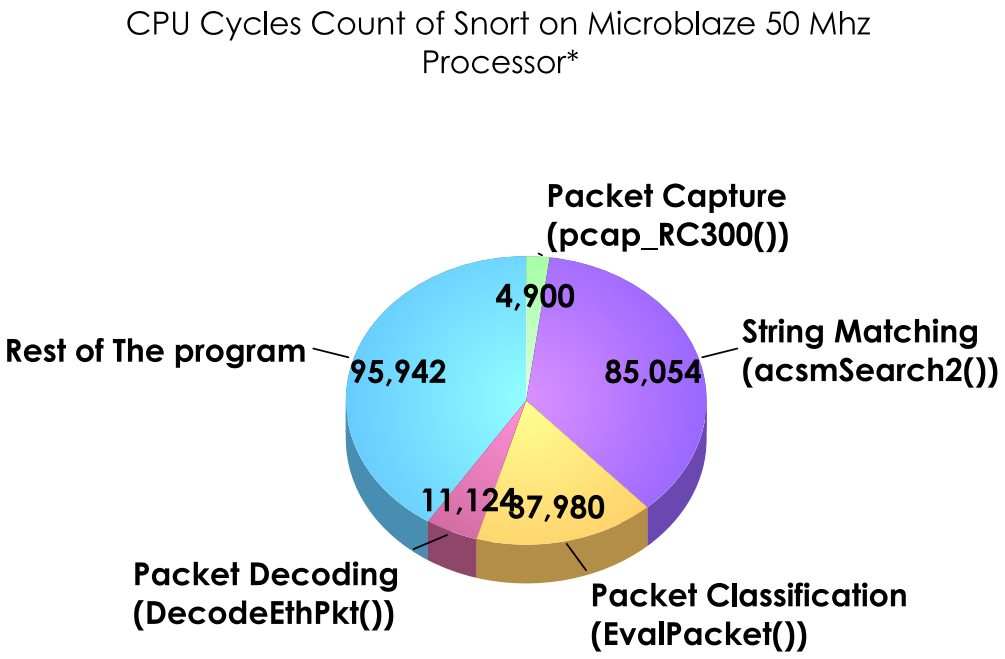


FIGURE 6.4: CPU cycles count of Snort on MicroBlaze

memory accesses only require tens of cycles. In contrast Snort on Intel Pentium platform takes hundreds of CPU cycles just for single memory access.

Summaries of synthesis and place and route reports are presented in order to indicate the utilisation of the FPGA device and the potential hardware still available for the provision of functional units. The synthesis results obtained for final design on a clock speed of 50 MHz are in figure 6.5.

```
Device Utilization Summary:
Number of DCMs:                      2 out of 12     16%
Number of External IOBs:           491 out of 824    59%
Number of MULT18X18s:                7 out of 144     4%
Number of Block RAM:                41 out of 144    28%
Number of occupied SLICES:        4092 out of 33792  14%
```

FIGURE 6.5: Synthesis result on Xilinx XC2V6000 -4 Virtex-II FPGA

This information shows that the current prototype on the FPGA only occupy 15% of FPGA area, so more hardware such as pattern matching were easily fitted easily into this design.

6.4 Testing and Evaluation of Pattern Matching Hardware Accelerator (MMU-Snort II and MMU-Snort III)

Only the performance test is carried out to determine the improvements of PMHA developed for Snort-based SB-NIDS prototype or MMU-Snort I on RC300 board.

6.4.1 Performance Test

The performance test involved memory test, throughput test and false positive test. In memory test the pattern memory space requirement is obtained and presented which later also compared with state of the art solution (Section 6.4.2). The throughput test is performed to identify the PMHA performance with/without application specific information. This test result is also compared with state of the art solutions (Section 6.4.2). False positive test involved determining false positive rate of Bloom filter. This included measuring the hash function effect on actual false positive rate. Finally, FPGA design space synthesis result is presented that shows the effect of efficient implementation of hash function technique to compute large number of hash function.

Memory Test

The memory is the main component of the PMHA. Hence, it influences the accelerator size and limits the throughput. Storing compactly all attack patterns is crucial which can be achieved by either memory efficient data structure or by compression algorithm. Instead of designing any new memory efficient algorithm a Bloom filter based pattern matching filtering approach is used which results in significantly less memory requirement as shown in the figure 6.6.

The Bloom filter approach resulted in compactly storing all 7876 patterns unique attack pattern in just 0.0302 MB (247.50 Kbits) of FPGA block RAM (BRAM). So a large number of patterns can be checked in packets with low latency access to FPGA local memory which would increase the overall throughput of the system. In comparison to the state machine based algorithm such as variants of Aho-Corasick that requires huge amount of runtime memory. This occurs as their memory requirement increases linearly with the number of characters in pattern set as shown in the figure 6.7.

This graph shows the linear increase of memory requirement of Aho-Corasick state machine with the number of characters in pattern database. As the new unique pattern added to the Snort database then it will result in increase of Aho-Corasick state machine memory size and will also increase the pattern search time.

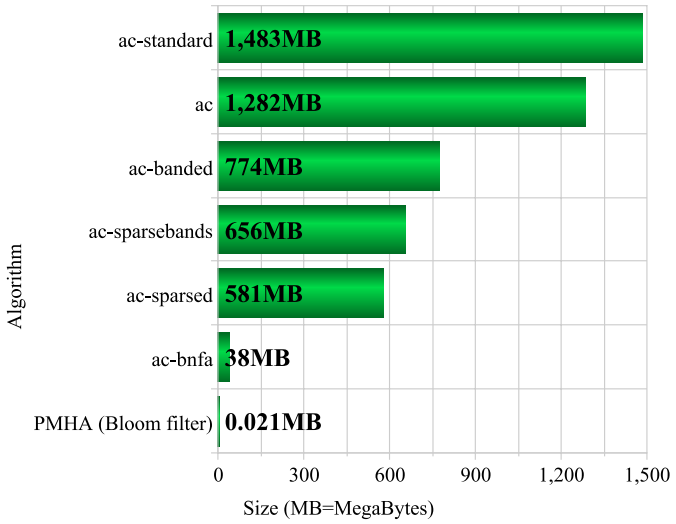


FIGURE 6.6: Aho-Corasick state machine and pattern matching hardware accelerator memory requirements

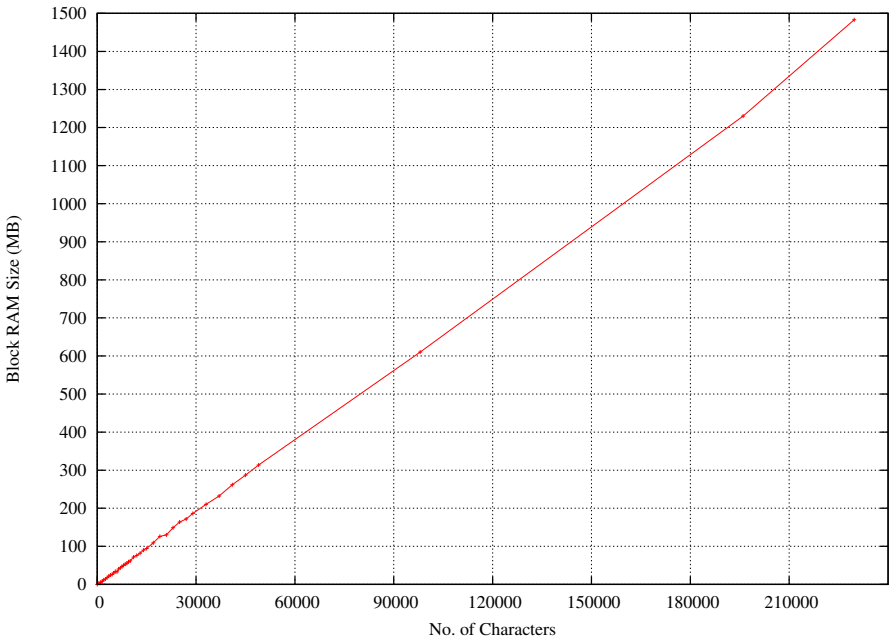


FIGURE 6.7: Aho-Corasick (ac-standard) state machine memory size (MB) for different character count

Clock Cycle count of MMU-Snort II

This test evaluates the PMHA performance in terms of number of clock cycles. The clock cycles count results are then used to compute the throughput of the PMHA. This system operates at 50 MHz clock frequency and tested with data from MIT Lincoln Lab website and shmoo group capture the flag website. Table 6.1 shows the number of clock cycles consumed by PMHA.

It is important to understand that the clock cycles count is heavily dependent on the

TABLE 6.1: Clock cycle count of Pattern Matching Hardware Accelerator (PMHA)

Pattern Size	Total Patterns	Packet Trace	Bytes Inspected	Clock Cycles (Without Rule Options)	Clock Cycles (With Rule Options)
< 16 bytes	3232	10 MBytes	597032	171683	148885
		15 MBytes	980590	326902	277148
16 to 31 bytes	2590	27 MBytes	656432	193815	172745
		30 MBytes	891343	237330	222835
32 to 64 bytes	1348	22 MBytes	234143	79323	71281
		47 MBytes	959121	413232	361932
> 64 Bytes	706	62 MBytes	414010	162859	153337
		35 MBytes	192121	80457	76848

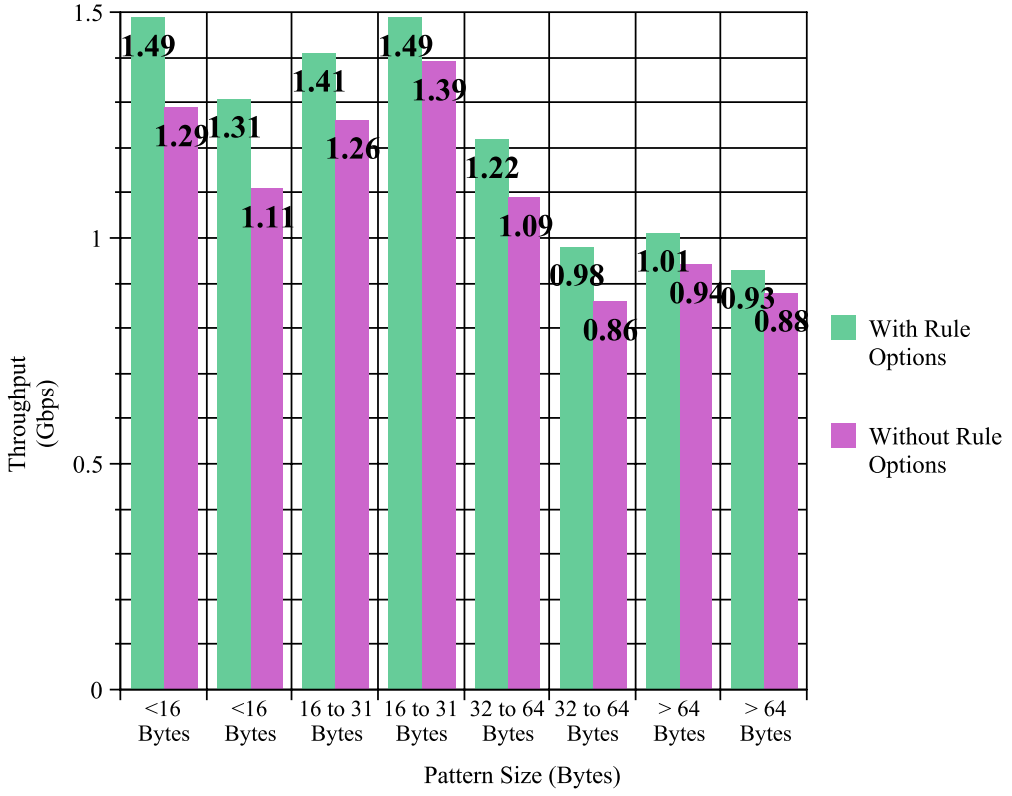


FIGURE 6.8: PMHA throughput at 50 MHz for the test results in Table 6.1

nature of test data which may have high number of patterns with variable length. Due to this reason the PMHA is tested each time for variable length patterns in order to get the accurate performance results. From the test results it is observed that for shorter size patterns (<16 bytes) search pattern matching algorithm consumes lesser number of clock cycles. However, the performance of pattern matching degraded for the longer size pattern (>64 bytes) search. The main reason of higher number of clock cycle for longer patterns is the hash value computation and false positive pruning which takes longer time for longer length pattern. This test result also revealed the effect of application specific knowledge (Snort rule options). The inclusion of these Snort rule

options specifies exactly the number of bytes of packet payload to search for patterns instead of the whole packet payload. This means less number of hash computation, Bloom filter lookup and false positive. The direct effect of all theses are lower number of clock cycles and hence better throughput. Figure 6.8 shows the effective throughput obtained for the clock cycles in table 6.1.

Pattern matching hardware throughput test results shows the best effective throughput 1.49 Gbps and the lowest throughput of 0.93 Gbps when PMHA operating at 50 MHz clock frequency. Also it can be seen clearly that the PMHA with Snort rule options provide higher throughput pattern matching. Another performance test is carried out with two MIT Lincoln lab data file and two defcon date file. This time the system is configured/programmed with all 7876 Snort patterns and with rule options. Figure 6.9 shows the throughput of pattern matching hardware operating at 50 MHz clock frequency.

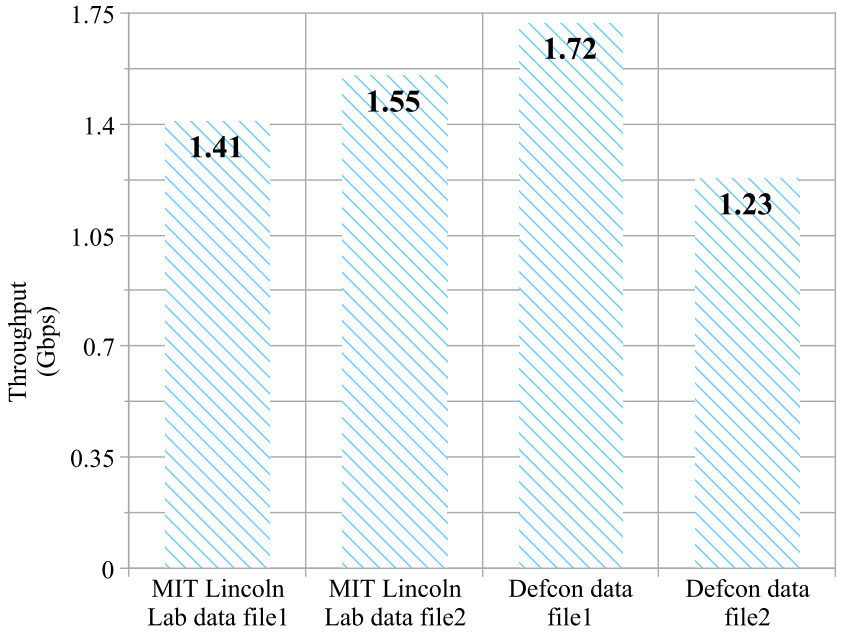


FIGURE 6.9: Pattern Matching Hardware Accelerator (PMHA) throughput with total 7876 patterns

It can be notice that when the PMHA programmed with all 7876 Snort pattern and with rule options provide much better throughput. The best throughput in this case is obtained as 1.72 Gbps.

Clock Cycle count of MMU-Snort III

Another clock cycle count test is carried out to evaluate the performance of optimised PMHA or MMU-Snort II (Section 5.4). This optimised hardware accelerator more efficiently prune the false positive results and search the longer size patterns ($>$ 64 bytes).

Table 6.2 contains the comparison of the clock cycle count of pattern matching hardware before optimisation (MMU-Snort II) and after optimisation (MMU-Snort III).

TABLE 6.2: Comparison of clock cycle count of PMHA before and after optimisation

Pattern Size	Total Patterns	Packet Trace	Bytes Inspected	Clock Cycles (Unoptimised)	Clock Cycles (Optimised)
< 16 bytes	3232	10 MBytes	597032	148885	135193
		15 MBytes	980590	277148	256932
16 to 31 bytes	2590	27 MBytes	656432	172745	169009
		30 MBytes	891343	222835	200017
32 to 64 bytes	1348	22 MBytes	234143	304169	261121
		47 MBytes	959121	217608	184678

It is observed from the test result that the MMU-Snort III improve the overall result by searching variable size patterns with lesser number of clock cycles. This is mainly due to efficient false positive pruning and faster optimised pattern matching for longer size patterns. Throughput calculated based on these clock cycles clearly shows the performance improvement as shown in figure 6.10.

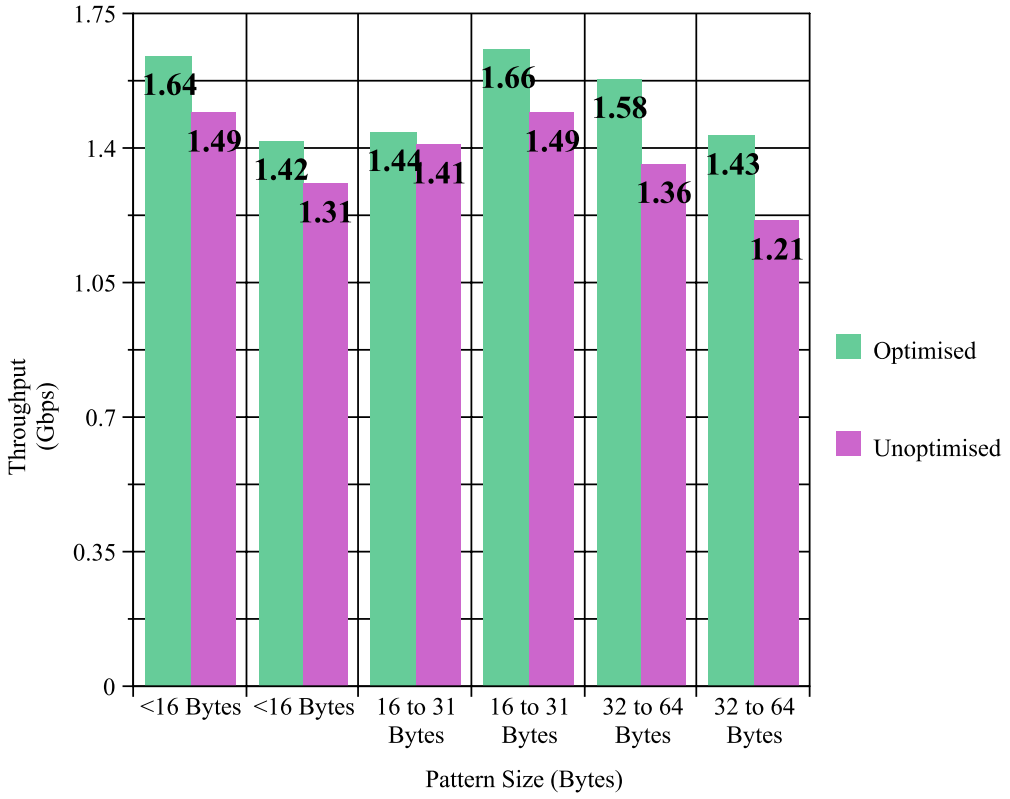


FIGURE 6.10: PMHA throughput comparison before and after optimisation

There are not anymore longer size pattern (> 64 bytes) in this test as these patterns has broken. Overall the throughput of PMHA shows the better performance for every size patterns. Another test is carried to measure the throughput of pattern matching

hardware after optimisation with total of 9150 Snort patterns which increases as a result of pattern breakup. This throughput is compared with the throughput obtained previously with 7876 patterns as shown in figure 6.11.

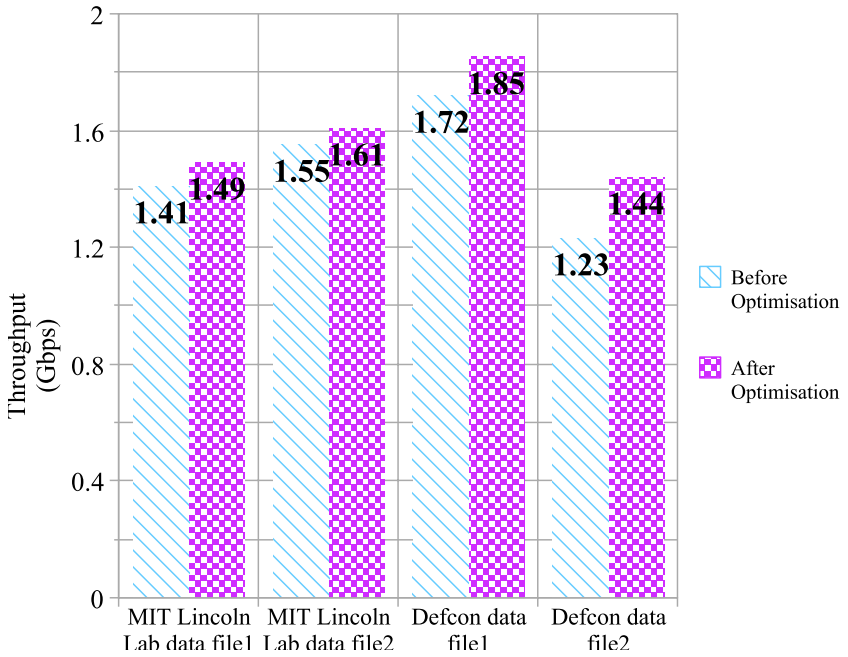


FIGURE 6.11: PMHA throughput comparison before and after optimisation

Even with the increase in number of pattern and false positive rate the optimised PMHA provide better pattern matching throughput. The best throughput observed in the optimised PMHA for all 9150 Snort pattern is 1.85 Gbps.

False Positive Test

Two hash functions *MD5* and *XOR-based* hardware hash function effect on Bloom filter false positive rate are compared. MD5 is picked up for comparison as it produced well distributed hash value. XOR-based hardware hash function is implemented in PMHA.

In order to test the effect of two different hash function on false positive rate a test program is written in JAVA using jdk 1.6.0. The test program is executed on a computer cluster at Manchester Metropolitan University for 48 hours. There were total of 7876 (n=7876) distinct patterns used to obtained false positive rate for different Bloom filter sizes. Bloom filter sizes in this test range from 2 KiloBytes to 10 KiloBytes (m=16384 to m=81920 number of bits). For each Bloom filter, the test program randomly selected the 90 % of patterns out of 7876 to insert/program into Bloom filters, and then the test

program query the Bloom filters for the rest of 10 % patterns. This process is repeated for 48 hours for each Bloom filter size for both hash functions. The final result shows the comparison of false positive rate of each Bloom filter vector for both hash functions and predicted false positive rate calculated with equation 5.1. Figure 6.12 shows the result summary.

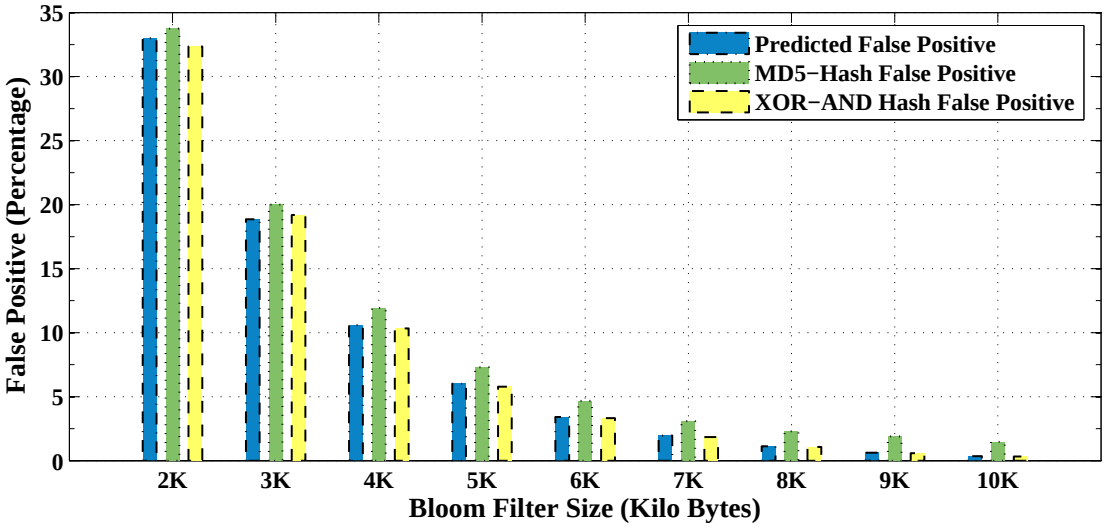


FIGURE 6.12: False Positive vs Bloom filter size

For each Bloom filter query, the XOR-based hash function produced lower false positive rates than MD5 hash function. Also XOR-based hash function produced better false positive rate than predicted false positive rate with the exception when Bloom filter size is set to 3 Kilobytes.

Another false positive test is carried out to analyse the practical false positive rate of Bloom filter lookup in pattern matching hardware. The 8 KiloBytes (m=65535 bits) Bloom filter programmed with 7876 number of patterns (n=7876) with a predicted false positive rate of 1.8390 % calculated from equation 5.1. Table 6.3 shows the false positive rate obtained for different network trace files.

TABLE 6.3: False positive rate of PMHA (MMU-SnortII) with 7876 patterns

Packet Trace	Total Data	Pattern Lookup	False Positives	False Positive Rate
MIT Lincoln Lab	50 MBytes	40050	537	1.3408 %
MIT Lincoln Lab	100 MBytes	52541	793	1.5092 %
MIT Lincoln Lab	250 MBytes	72012	1116	1.5497 %
Defcon	20 MBytes	20150	282	1.3995 %
Defcon	12 MBytes	21020	336	1.5984 %

The false positive rate obtained from the test is lower and better than predicted false positive rate. The best case false positive rate is 1.34 % which is approximately 1.37 times better than predicted false positive rate.

The Snort pattern number has increased due to the pattern breakup so the optimised PMHA is also tested for practical false positive rate of Bloom filter. The optimised pattern matching hardware acceleration has 8 KiloBytes (m=65535 bits) Bloom filter programmed with 9150 number of patterns (n=9150) with a predicted false positive rate of 3.1667 % calculated from equation 5.1. Table 6.3 shows the false positive rate obtained for different network trace files.

TABLE 6.4: False positive rate of PMHA (MMU-Snort III) with 9150 patterns

Packet Trace	Total Data	Pattern Lookup	False Positives	False Positive Rate
MIT Lincoln Lab	50 MBytes	40050	1069	2.6691 %
MIT Lincoln Lab	100 MBytes	52541	1508	2.8701 %
MIT Lincoln Lab	250 MBytes	72012	2102	2.9189 %
Defcon	20 MBytes	20150	556	2.7593 %
Defcon	12 MBytes	21020	621	2.9543 %

The false positive rate obtained from the test is lower and better than predicted false positive rate. The best case false positive rate is 2.66 % which is approximately 1.18 times better than predicted false positive rate.

FPGA Synthesis Results

Six pipelined hash functions are implemented separately to identify the FPGA design space requirement for computing six hash values in parallel. Summary of the FPGA synthesis results is shown in figure 6.13.

Device Utilisation Summary:

Number of DCMs:	3	out of	12	25%
Number of External IOBs:	324	out of	824	39%
Number of MULTI8X18s:	7	out of	144	4%
Number of Block RAM:	38	out of	144	26%
Number of occupied SLICES:	13234	out of	33792	39%

FIGURE 6.13: Synthesis result of six hash module on Xilinx XC2V6000 -4 Virtex-II FPGA

The synthesis result of implementation which can compute N number of hash function using only two hash is shown in figure 6.14.

This implementation has two hardware hash modules and one 2-to-N hash modules (Section 5.3.3) to generate N number of hash values using two hash values. The 2-to-N hash module is implemented in the PMHA to reduce the FPGA design space requirement of hash function modules. This implementation use less block RAM and occupy less than 17 % of FPGA slices than the six hash module implementation. The final synthesis results of PMHA integrated with Snort port on RC300 is shown in figure 6.15.

Device Utilisation Summary:				
Number of DCMs:	2	out of	12	16%
Number of External IOBs:	290	out of	824	35%
Number of MULTI8X18s:	7	out of	144	4%
Number of Block RAM:	24	out of	144	16%
Number of occupied SLICES:	7591	out of	33792	22%

FIGURE 6.14: Synthesis result of two hash module and 2-to-N hash module on Xilinx XC2V6000 -4 Virtex-II FPGA

Device Utilisation Summary:				
Number of DCMs:	6	out of	12	50%
Number of External IOBs:	523	out of	824	63%
Number of MULTI8X18s:	7	out of	144	4%
Number of Block RAM:	121	out of	144	84%
Number of occupied SLICES:	27120	out of	33792	80%

FIGURE 6.15: Synthesis result of full SB-NIDS prototype (MMU-Snort III) on Xilinx XC2V6000 -4 Virtex-II FPGA

This synthesis result is obtained for a full prototype system which consists of PMHA, packet capture hardware accelerator and Decision Engine hardware accelerator integrated with Snort port on RC300 board. This information shows that the full prototype on the FPGA occupy 80 % of FPGA area, so more hardware such as stateful packet inspection hardware accelerator can still fitted into this design.

6.4.2 Comparison with Previous Work

In comparison to other state of the art this PMHA is memory efficient as it provides high speed pattern lookup in Bloom filter programmed in FPGA BRAM. This comparison is shown in table 6.5.

TABLE 6.5: Pattern Matching Hardware Accelerator (PMHA) Memory Size (Kbits)

Related Work	Snort Patterns	FPGA	BRAM Size (Kbits)	BRAM Used (Kbits)
Gokhale et al [87]	1500-2000	Xilinx Virtex 1000 XCV1000	128	23.04
Sarang et al [81]	1434	Xilinx Virtex XCV2000E	640	140
Attig & Lockwood [94]	2464	Xilinx Virtex XCV2000E	640	568
PMHA (MMU-Snort III)	9150	Xilinx Virtex-II XC2V6000	324	180

The performance of pattern matching hardware is also compared with state of the art pattern matching hardware solution. The performance is compared in terms of throughput and design space/area of FPGA. Table 6.6 shows the throughput of different state of

the pattern matching hardware and pattern matching hardware architecture on RC300 board.

TABLE 6.6: Pattern matching hardware architecture on FPGA

Related Work	Snort Patterns	Throughput
Gokhale et al [87]	1500-2000	2.0 Gbps
Yusuf et al [85]	74	2.8 Gbps
Attig & Lockwood [94]	2464	2.5 Gbps
PMHA (MMU-Snort III)	9150	1.85 Gbps

The throughput recorded for the proposed pattern matching hardware on RC300 board is lower that other similar state of the art pattern matching hardware architecture. However, the lower throughout is due to operating frequency of 50 MHz which is maximum possible achievable with MicroBlaze on RC300 board. Another reason of lower throughput is the number of rules the system is tested with which in compare to other systems is much higher than other pattern matching hardware architecture.

6.5 Chapter Summary

This chapter presented the detail testing and evaluation of the SB-NIDS prototype and hardware accelerator. This chapter began with the introduction of testbed network topology. This is followed by the testing and evaluation of novel SB-NIDS prototype. The testing and evaluation involved functional and performance test. The functional test is carried out to make sure the SB-NIDS work error free. The performance test identifies the bottleneck and improvement of porting a system on high performance computing platform. Finally, the PMHA testing and evaluation is presented which involved mainly performance testing and comparison with state of the art pattern matching hardware solution.

Chapter 7

Conclusion and Future Work

The objective of this thesis is the optimisation of Signature-based Network Intrusion Detection System (SB-NIDS) packet analysis speed using the high performance embedded processing platform.

7.1 Chapter Summary

Low speed packet analysis of SB-NIDS becomes bottleneck on high data rate network. Due to this reason various SB-NIDS solutions has been proposed. Some of these solutions use high performance 6processing technology to optimise the SB-NIDS. These technologies mainly include cluster of processors and embedded processing platforms. Cluster of processors are expensive in terms of cost. Their maintenance cost are also very high. In comparison embedded processing technologies are compact size. They offer high performance processing and easy to deploy for network monitoring and surveillance. Their maintenance cost is also lower in comparison to cluster of processors. Therefore, the embedded processing technology is viable and so chosen to develop and optimise SB-NIDS. The embedded technology used is FPFA-MicroBlaze based hybrid hardware-software processing platform. It is tightly coupled hardware architecture with two gigabit Ethernet network interfaces. It also provide ways to offload processing from processor to hardware and has multiple processing cores. Due to these features it is an ideal platform for SB-NIDS prototyping and optimisation which is the subject of this thesis research. In summary, *Introduction* chapter 1 introduces this research, contains the problem statement, aims and objectives, outcome and contribution of this research.

Background chapter 2, is the study of the core concepts related to this research. This includes an overview of network security issues and network defence technologies. The

main emphasise is on SB-NIDS network defence technology to highlight limitations and issues concerning this technology which is the main motivation of this research.

Survey and Related Work chapter 3 is the detail survey of the related work to this thesis. This survey is organised into two parts: i) High performance SB-NIDS architecture and ii) Pattern matching for SB-NIDS. The first part is dedicated to the state of the art SB-NIDS solutions proposed for performing high speed packet analysis. These state of the art SB-NIDS solutions are implemented using cluster of processors and embedded processing hardware architecture. The conclusion of this part of survey found that SB-NIDS implementation using embedded processing platforms is viable solution as compare to cluster of processors. They are lower cost and offer ease of deployment, maintenance and further development. The second part is dedicated to state of the art pattern matching algorithms and hardware architecture for SB-NIDS. The state of the art pattern matching is logically organised into three categories: i) SB-NIDS specific pattern matching, ii) packet filtering technique and iii) High performance pattern matching hardware architecture. SB-NIDS specific pattern matching algorithms are hybrid pattern matching algorithms developed by combining state machine and skip table search technique. Packet filtering techniques are algorithms for filtering as much network traffic as possible in order to reduce the amount of traffic to be sent for analysis in SB-NIDS. Pattern matching is implemented using high performance embedded processing architecture. Two high performance processing platforms used to implement pattern matching are *Network processors* and *Field Programmable Gate Arrays* (FPGAs). In all implementations FPGA based designs provide better and high throughout pattern matching solutions. The average throughput observed is well over 1.0 Gbps.

Proposed System Architecture chapter 4 contains the study to identify the development challenges and requirements. To pursue this effort SB-NIDS software package Snort and high performance embedded processing hardware architecture HandelC-MicroBlaze is studied and discussed. This study helped to understand the SB-NIDS internal working and processing architecture. It also help identifying performance issue and formulating plan for SB-NIDS development and performance optimisation as explained in chapter 5.

Design and Implementation chapter 5 contains the detail system design and implementation which is carried out in three stages resulted in three system prototypes: i) Manchester Metropolitan University (MMU-Snort I), ii) MMU-Snort II and iii) MMU-Snort III. MMU-Snort I is the novel SB-NIDS prototype based on Snort developed on Mentor Graphics RC300 board. The prototyping involved Snort internal architecture restructuring and mapping/porting to HandelC-MicroBlaze based architecture. This prototyped executes only on single MicroBlaze core with two HandelC hardware accelerator units: Packet Capture Hardware Accelerator (PCHA) and Decision Engine

Hardware Accelerator (DEHA). PCHA is the high speed and low latency packet capturing facility that captures packets directly from one of the gigabit Ethernet interface on RC300 board. DEHA is the replacement of Snort's Decision Engine component. Its job is to send the detection results through second gigabit Ethernet interface on RC300 board for reporting to Network Administrator. MMU-Snort I is the first ever such SB-NIDS that utilises hybrid hardware-software embedded processing platform power. It is faster when performance is compared with the Snort on general purpose processor. MMU-Snort II is the Pattern Matching Hardware Accelerator (PMHA). PMHA is implemented with Bloom filter based pattern search approach. It is improved and better pattern search algorithm design than other Bloom filter based pattern matching implementations. PMHA novel features are integrated pattern matching hardware with Snort port on MicroBlaze, compact storage of largest number of attack patterns (9150 patterns) in FPGA local memory, and lower number of pattern lookup in Bloom filter achieved by using Snort application specific knowledge (Snort rule options). MMU-Snort III is the PMHA optimised and extended version. This optimisation involved an algorithmic and architectural improvement for efficient pattern pruning and faster longer size patterns ($>$ 64 bytes) search. Efficient patterns pruning technique is achieved by utilising Snort attack rules unique identification number (Rule ID). Using Rule ID for pattern pruning lowers the number of pattern comparisons and resulted in faster pruning. For faster longer size pattern ($>$ 64 bytes) search, all over 64 bytes pattern is broken down. If the first part of the longer pattern is match with packet data after comparing using pattern comparator then the second part is checked in Bloom filter, otherwise the search is stop. Efficient pruning and faster longer pattern search result in increase of pattern matching throughput. Finally, the testing of all prototypes are carried out which is explained in *Results and Analysis* chapter 7.

Results and Analysis chapter 7 contains the testing and evaluation of all three prototypes. MMU-Snort I is tested with publicly available test data (MIT Lincoln Lab and Defcon Shmoo group) to identify any functional issues and performance improvement. Functional test results showed the correction detection result of network packet analysis. The performance result of MMU-Snort I on hybrid hardware-software embedded processing platform is compared with the Snort performance on general purpose processor on personal computer. It was concluded that Snort port on RC300 board (MMU-Snort I) performance is 1.7 times better than original unmodified Snort software package on general purpose processor. This difference is due to the tightly coupled hardware architecture of embedded processing platform. Also PCHA and DEHA also improved the overall speed of the packet analysis speed. The test results also indicated the computationally intensive operations of Snort. These include pattern matching algorithm, Stateful Packet Inspection (SPI) and packet classification. MMU-Snort II which

is the PMHA integrated with MMU-Snort I is tested with publicly available test data to identify performance improvement. The test results showed that at 50 MHz operating frequency the highest and lowest throughput of 1.72 Gbps and 1.23 Gbps respectively when 7876 Snort attack pattern is searched in packet. Another throughput test for varying length patterns indicated the lower throughput for pattern matching for longer size pattern (> 64 bytes). The throughout dropped to 0.86 Gbps in this case. Further throughput test is conducted to evaluate application specific knowledge inclusion in search algorithm. The throughout result clearly indicated better results for pattern matching algorithm with application specific knowledge (Snort rule options) which is one of the major contribution of this research. The false positive test results also showed the better performance of MMU-Snort II. The best false positive rate found was 1.34 % which is 1.37 times lower than predicted rate of 1.83 %. MMU-Snort III which is the final prototype with optimised PMHA is also evaluated with some series of test. The optimised PMHA offers efficient pattern pruning and faster longer pattern search. The throughput results with 9150 attack patterns at 50 MHz operating frequency shows the increase in best case throughout from 1.72 Gbps to 1.85 Gbps. The lowest case throughput also increased from 1.23 Gbps to 1.41 Gbps. This PMHA also supports highest number of 9150 Snort attack patterns. All these patterns are compactly stored for quick lookup in 8 KB (64 Kbits) of FPGA Block Random Access Memory (BRAM). Even with such a high number of attacks pattern FPGA synthesis result summary of PMHA shows only 180 Kbits of FPGA BRAM usage. The final prototype MMU-Snort III FPGA synthesis result summary shows the whole design occupy 80 % of FPGA logic resources and 84 % of BRAM (272 Kbits). MMU-Snort III comprises of PMHA, PCHA and DEHA integrated with Snort port executing on MicroBlaze. The test results also compared with the closely related work and appeared to have lower performance in terms of throughput. Main reason is the lower operating frequency achievable on this grade of FPGA with MicroBlaze. In comparison the FPGA used for pattern matching implementation in closely related work are higher grade and so support higher frequency and pattern matching throughput. Another reasons for lower operating frequency is the HandelC hardware description language. It requires extensive source code level refinement to reduce the hardware design complexity and to synthesise efficiently at higher operating frequency. In summary, HandelC is slower than other lower level Hardware description language like Verilog. It also demands extensive and time consuming source code refinement in order to achieve modest operating frequency.

7.2 Overall Conclusion

The primary objective of this thesis is the development of optimised SB-NIDS design which has been achieved and advanced the state of the art. The novel MMU-Snort III developed on tightly coupled hybrid hardware-software embedded processing architecture is the final version of SB-NIDS. It consists of three HandelC hardware accelerators: PCHA, DEHA, PMHA, while the rest of the Snort port executes on single MicroBlaze core. PCHA provides low latency and fast packet capturing facility. DEHA is the lighter version and replacement of Snort's Decision Engine component which provides fast intrusion analysis result reporting. PMHA is the high speed pattern matching solution for faster attack pattern search in packet. Series of test results shows improved overall performance indicating higher throughput and lower false positive rate. However, when throughput compared with closely related state of the art work it is lower than most them, but offers well above and one of the highest number of 9150 attack pattern search which also compactly stored in FPGA BRAM. FPGA device utilisation summary shows the MMU-Snort III prototype utilised 80 % of FPGA logic resources and 84 % of FPGA BRAM leaving very limited resources for further optimisation.

Let suppose MMU-Snort III design has migrated to high end Xilinx Virtex-7 FPGA and assume final FPGA device summary result shows only 50 % of FPGA logic resources and FPGA BRAM usage providing opportunity for further optimisation. What could be the next possible optimisation target? There are few attractive choices but the two most useful candidates are offloading of Transmission Control Protocol (TCP) packet analysis or Stateful Packet Inspection (SPI) from MicroBlaze to FPGA and exploitation of multiple processing resources to increase the overall efficiency of packet processing. SPI demands fast packet processing and frequent data memory access. It involve computing hash values on key packet header values (IP addresses and ports) of every incoming packet and using hash value it lookup on TCP connection table which normally stores in application memory or Random Access Memory (RAM) in order to determine packet association with established connection. Packet with the same IP address and ports are hashed to the same memory location and in this way packet can be identified as a part of connection. Offloading SPI to FPGA would benefit FPGA fast memory access and parallel processing feature. This would significantly increase overall packet analysis speed. Also SPI is comparatively simple process than pattern matching and so SPI hardware accelerator would only consume very limited FPGA logic resources. For example, hash value computation on FPGA is a straight forward operation which can easily be computed on key packet header values (IP address and port number) in around 5-10 clock cycles, while another 2 clock cycles for accessing off-chip memory on

this hybrid hardware-software platform to access TCP connection table for retrieving or storing connection information.

Another option which is now going to be discussed is the usage of parallel processing available in the form of multiple MicroBlaze core. The biggest question that arises is which part of SB-NIDS and how to execute these parts on multiple MicroBlaze core. According to SB-NIDS architecture and execution analysis it was identified that there are some SB-NIDS components that are independently process the data without relying on any other components. Each of these components can be assigned a separate Microblaze core for processing. These SB-NIDS components are SB-NIDS Preprocessor. Some of these are Simple Mail Transfer Protocol (SMTP) Preprocessor, Domain Name System (DNS) Preprocessor, Hypertext Transfer Protocol (HTTP) Preprocessor, File Transfer Protocol (FTP) Preprocessor and Telnet Preprocessor. These Preprocessor perform packet analysis only on particular part of decoded packet data. They simply take input the decoded data, process them and give the output which is the detection result summary. Assigning these Preprocessor a separate MicroBlaze core would certainly increase the overall efficiency of packet processing. The implementation of such effort would be highly complex. It would require major architectural changes of the current MMMU-Snort III prototype.

There are other limitation exists in final MMU-Snort III prototype. These are the limitations that could also be the candidate for further research and development. These now discuss in the following section 7.3.

7.3 Limitations and Future Directions

This section will briefly examine further future research directions resulting from the work presented in this thesis.

7.3.1 Regular Expression Search

Mitra et al and Brodie et al presented the FPGA hardware architecture to perform regular expression based search [102, 103]. These hardwares are designed to search the malicious patterns in network packets up to gigabit rate throughput. Mitra et al hardware architecture addressed the issue of regular expression search of Snort attack rules. This idea can be applied to the PMHA for Snort presented in this thesis. This will also enable the first full Snort port on hybrid hardware-software processing to perform not only ordinary pattern matching in FPGA but also a regular expression search which is an integral part of pattern search of Snort.

The regular expression search can be implemented using one of the same approaches that have been applied consistently in previous work on ordinary pattern matching. Implementing them in FPGA is rather challenging due to limited amount of memory and design space. One of the widely used approach is state machine implementation where regular expression based search can be based on and search is carried out in parallel using FPGA parallel logic resources as proposed by Mitra et al. Bloom filter based search approach can also be applied for regular expression search in which characters that are specifically specified in regular expression are check in packet data using Bloom filter. In case all specified characters in a pattern is found then in next step wild card character search is carried out either with brute force searching or using state machine based approach. Whatever the type of search technique is applied to regular expression search in FPGA it should result in fast and high throughput solution provided it is implemented efficiently so can be synthesized at high clock rate.

7.3.2 Non-Interruptible Update

In order for SB-NIDS to be effective it needs to have updated attack signature list. The signature update requires copying of new signatures to SB-NIDS files and restarting a SB-NIDS software package. During this process SB-NIDS does not able to inspect any network traffic leaving large number of packets to enter into network without inspection. An alternate is to have multiple SB-NIDS deployment on a network while one is being updated then other inspects the network traffic. This is rather expensive solution and the best option is to have non-interruptible update facility where the same SB-NIDS continued the network traffic inspection while signature database is being updated with new signatures.

Non-interruptible update facility is difficult to implement in SB-NIDS software. On hybrid hardware-software processing platform this is further challenging as part of signatures known as patterns stored in RC300 board off-chip Synchronous Dynamic Random Access Memory (SDRAM) as well as in FPGA BRAM. Rest of the signature parts are stored in MicroBlaze memory. Viable solution on this platform is FPGA hardware functional unit which will promptly update the signature in both SDRAM and FPGA BRAM. Also fundamental changes in MMU-Snort III would also be required, such as Bloom filter needs to change to counting Bloom filter which allows pattern addition and deletion. MicroBlaze memory or Static Random Access Memory (SRAM) is tricky to perform signature update. Only way to do this successfully is to create a copy of updated signatures data structure in memory and overwrite it to the original signature memory. This is the complicated task and may cause SB-NIDS to stop search for a short interval of time, possibly few seconds which would result few packets un-inspected.

7.3.3 Packet filtering

In chapter 3 3 some state of the art work are discussed that applied the packet filtering techniques to SB-NIDS in order to optimised NIDS pattern matching algorithm performance. Packet filtering can also be applied to the MMU-Snort III as well. Packet filtering implementation is more appropriate and effective. For example, a filtering system quickly checks the encapsulated protocols in packets and filter them if no network/application/operating system is processing such packet. These packets do not required analysis and cannot harm the network.

7.4 Final Comments

MMU-SnortIII is a complete SB-NIDS prototype developed on high performance hybrid hardware-software embedded processing platform using state of the art Snort SB-NIDS software package. Unlike previous state of the art solutions that have limited threat detection features this SB-NIDS prototype has one of the effective detection facility provided by Snort. It effectively analyses many different types of protocol (Layer-3 to Layer-7 of Open System Interconnection (OSI) layer) and able to detect many new attacks due to regular signature database updates release. Also the processing platform high performance processing facility has been exploited during prototype development which resulted in improvement of packet analysis speed. A novel hardware accelerator of pattern matching is also developed that boosted the packet analysis speed. The novel feature of this hardware accelerator is the first ever integrated PMHA to full SB-NIDS software package. Another novel feature which proved successful in improving the packet analysis speed is the use of application specific knowledge (Snort rule options). This reduces the number of pattern search and so lower the search computation time. It also supports one of the largest databases of attack patterns (9150) search which it stores compactly in FPGA BRAM for quick pattern search. It also offers a decent throughput of 1.85 Gbps operating at 50 MHz of clock frequency. The lower frequency is the limitation imposed by the development platform otherwise this hardware accelerator itself without MicroBlaze on the same grade FPGA can be synthesized at 80 MHz.

The final MMMU-Snort III prototyped is far from perfect and has limitations (Section 7.3). One of the concerns is related to the development platform which has lower grade Xilinx Virtex-II FPGA and does not support higher operating frequency and limited FPGA design/space. MMU-Snort III performance can be boost up to 10 times if design is migrated the high end Xilinx Virtex-7 FPGA which would easily able to synthesise the design over 650 MHz operating frequency. In summary, MMU-Snort III

on hybrid hardware-software processing platform is found to be a well suited platform for SB-NIDS performance optimisation and this prototype can serve as a platform for further research and development for SB-NIDS performance optimisation.

Bibliography

[1] Andrew Baker Jay Beale and Joel Esler. *Snort IDS and IPS toolkit*, chapter 5 (Inner Workings), pages 177–178. Syngress press, 2007.

[2] Intel Cooperation. Supra-linear packet processing performance with intel multi-core processors. Published online by Intel Cooperation (8-pages), 2006. URL `http://download.intel.com/technology/advanced_comm/31156601.pdf`. Last access: 24 Nov. 2009.

[3] A light weight network intrusion detection system. WWW page, 1998. URL `http://www.snort.com`. Last accessed: 23/11/2009.

[4] IEEE. Ieee launches next generation of high-rate ethernet with new ieee 802.3ba standard. WWW page, May 2010. URL `http://standards.ieee.org/news/2010/ratification8023ba.html`. Last accessed: 21/12/10.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[6] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008. ISSN 1042-9832.

[7] British Broadcasting Corporation (BBC). Online scams target the wealthy. WWW page, Nov 2006. URL `news.bbc.co.uk/news/1/hi/technology/6135246.stm`. Last accessed: 21/01/10.

[8] Dan Lothian of Cable News Network (CNN). Authorities investigate online 'hitman' scams. WWW page, June 2007. URL `www.cnn.com/2007/US/06/18/lothian.cybercrime/index.html`. Last accessed: 21/01/10.

[9] British Broadcasting Corporation (BBC). Bank loses $1.1m to online fraud. WWW page, Jan 2007. URL `news.bbc.co.uk/news/1/hi/business/6279561.stm`. Last accessed: 21/01/10.

[10] British Broadcasting Corporation (BBC). Google 'may pull out of china after gmail cyber attack'. WWW page, Jan 2010. URL `news.bbc.co.uk/news/1/hi/business/8455712.stm`. Last accessed: 21/01/10.

[11] thisisexeter. Computer virus shuts down exeter university system. WWW page, Jan 2010. URL `www.thisisexter.co.uk/news/hacker-shuts-university/article-1729355-detail/article.html`. Last accessed: 21/01/10.

[12] Ross Anderson. *Security Engineering: A guide to building dependable distributed systems*. John Wiley and Sons, Inc, NY, USA, 1st edition edition, 2001.

[13] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. A framework for classifying denial of service attacks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 99–110, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4.

[14] CERT ©Coordination Center. Trends in denial of service attacks technology, Oct 2001. URL `http://www.cert.org/archive/pdf/DoS_trends.pdf`. Last accessed: 27/01/10.

[15] W. Eddy. Memorandum on tcp syn flooding attacks and common mitigations. WWW page, Aug 2007. URL `http://tools.ietf.org/html/rfc4987`. Last accessed: 26/01/10.

[16] Paul A. Watson. Slipping in the window: Tcp reset attacks. Technical report, terrorist.net, Dec 2003. URL `http://www.bandwidthco.com/whitepapers/netforensics/tcpip/TCP%20Reset%%20Attacks.pdf`. Last accessed:06/02/10.

[17] Computer Emergency Response Team (CERT). Cert advisory ca-1998-01 smurf ip denial-of-service attacks. WWW page, Jan 1998. URL `http://www.cert.org/advisories/CA-1998-01.html`. Last accessed: 07/02/10.

[18] Computer Emergency Response Team (CERT). Cert advisory ca-1998-01 smurf ip denial-of-service attacks. WWW page, Dec 1996. URL `http://www.cert.org/advisories/CA-1996-26.html`. Last accessed: 07/02/10.

[19] D. Senie. Changing the default for directed broadcasts in routers. WWW page, Aug 1999. URL `http://www.ietf.org/rfc/rfc2644.txt`. Last accessed: 07/02/10.

[20] Computer Emergency Response Team (CERT). Cert advisory ca-1996-01 udp port denial-of-service attack. WWW page, Feb 1996. URL `http://www.cert.org/advisories/CA-1996-01.html`. Last accessed: 07/02/10.

[21] Computer Emergency Response Team (CERT). Cert advisory ca-1997-28 ip denial-of-service attack. WWW page, Dec 1997. URL `http://www.cert.org/advisories/CA-1997-28.html`. Last accessed: 08/02/10.

[22] Office of the chief information commissioner. WWW page, 2010. URL `http://www.doecirc.energy.gov/aboutus.html`. Last accessed: 21/12/2010.

[23] Paul J. Criscuolo. Distributed denial of service trin00,tribe flood network,tribe flood network 2000,and stacheldraht ciac-2310. WWW page, Feb 2000. URL `http://doecirc.energy.gov/documents/CIRC-2319_Distributed_Denial_of%_Service.pdf`. Last accessed: 08/02/10.

[24] British Broadcasting Corporation (BBC). Yahoo attack exposes web weakness. WWW page, Feb 2000. URL `news.bbc.co.uk/news/1/hi/sci/tech/635444.stm`. Last accessed: 08/02/10.

[25] Patrick Mcdaniel Kevin Butler, Toni Farley and Jennifer Rexford. A survery of bgp security. Technical report, AT&T Labs-Research, April 2005. URL `www.patrickmcdaniel.org/pubs/td-5ugj33.pdf`. Last accessed: 10/02/10.

[26] SANS Institute InfoSec Reading Room. Icmp attacks illustrated. WWW page, 2001. URL `http://www.sans.org/reading_room/whitepapers/threats/icmp_attacks_illustrated_477?show=477.php&cat=threats`. Last accessed: 06/02/10.

[27] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computer & Security*, 24(1):31–43, Feb 2005.

[28] B. Martin, M. Brown, A. paller, and S. Christy. 2009 cwe/sans top 25 most dangerous programming errors. WWW page, Oct 2009. URL `http://cwe.mitre.org/top25/archive/2009/2009_cwe_sans_top_25.pdf`. Last accessed: 12/02/10.

[29] US Department of Homeland Security SRI International Identity Theft Technology Council and the Anti-Phishing Working Group. The crimeware landscape: Malware, phishing, identity theft and beyond. WWW page, Oct 2006. URL `http://www.antiphishing.org/reports/APWG_CrimewareReport.pdf`. Last accessed: 12/02/10.

[30] Ken Dunham. *Mobile Malware Attacks and Defense*. Syngress Publishing, 2008. ISBN 1597492981,9781597492980.

[31] Frederick B. Cohen. *Computer viruses*. PhD thesis, University of Southern California, Los Angeles, CA, USA, 1986.

[32] S.R Subramanya and N Lakshminarasimhan. Computer viruses. In *IEEE potentials*, pages 16–19, Oct/Nov 2001.

[33] Samuel Mc Innes Bechard, Anita K. Jones, and Robert S. Sielken. Computer system intrusion detection: A survey. Technical report, University of Virginia, 1999. URL `http://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/IntrusionDetection_jones-sielken-survey-v11.pdf`. Last accessed: 22 Feb. 2010.

[34] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, pages 2435 – 2463, 1999.

[35] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P Anderson Co., Fort Washington, Pennsylvania,USA, April 1980. URL `http://csrc.nist.gov/publications/history/ande80.pdf`. Last accessed: 23 Feb. 2010.

[36] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM. ISBN 0-89791-732-4.

[37] Ossec: An open source host-based intrusion detection system. WWW page, 2004. URL `http://www.ossec.net/`. Last accessed: 23/12/2010.

[38] Bro: Unix-based network intrusion detection system. WWW page, 1998. URL `http://www.bro-ids.com`. Last accessed: 23/11/2009.

[39] Cisco network intrusion prevention system 4200 series. WWW page, 2009. URL `http://www.cisco.com/en/US/prod/collateral/vpndevc/ps5729/ps5713/ps4077/ps9157/product_data_sheet09186a008014873c.pdf`. Last accessed: 23/12/2010.

[40] Ibm proventia intrusion prevention system. WWW page, 2007. URL `http://www.ibm.com/ru/services/iss/pdf/ibmproventianetworkipscomparison.pdf`. Last accessed: 23/12/2010.

[41] Sourcefire ids sensor. WWW page, 2009. URL `http://www.sourcefire.com/resources/sourcefire-3d9900-sensor`. Last accessed: 23/12/2010.

[42] Tippingpoint n platform. WWW page, 2009. URL `http://h10163.www1.hp.com/pdf/resources/datasheets/401221-002_N-PlatformTechSpecs.pdf`. Last accessed: 23/12/2010.

[43] Mike Schiffman. *Building Open Source Network Security Tools: Components and Techniques*. John Wiley and Sons, 2002. ISBN 0471205443.

[44] Nitesh Dhanjani and Justin Clarke. *Network Security Tools*. O'Reilly Media, 2005. ISBN 0596007949.

[45] Chris McNab. *Network Security Assessment: Know Your Network*. O'Reilly Media, 2004. ISBN 059600611Z.

[46] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, 2009.

[47] Javier Verdú, Jorge García, Mario Nemirovsky, and Mateo Valero. The impact of traffic aggregation on the memory performance of networking applications. In *MEDEA '04: Proceedings of the 2004 workshop on MEmory performance*, pages 57–62, New York, NY, USA, 2004. ACM.

[48] Javier Verdú, Jorge Garcí, Mario Nemirovsky, and Mateo Valero. Architectural impact of stateful networking applications. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 11–18, New York, NY, USA, 2005. ACM. ISBN 1-59593-082-5.

[49] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 203–214, New York, NY, USA, 1998. ACM. ISBN 1-58113-003-1.

[50] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 147–160, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6.

[51] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998. ISSN 0146-4833.

[52] Anthony J. Mcauley and Paul Francis. Fast routing table lookup using cams. In *IEEE INFOCOM*, pages 1382–1391, 1993.

[53] Florin Baboescu, Sumeet Singh, and George Varghese. Packet classification for core routers: Is there an alternative to cams? In *INFOCOM*, 2003.

[54] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. ISSN 0001-0782.

[55] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. ISSN 0001-0782.

[56] Stephen M. Specht and Ruby B. Lee. Distributed denial of service: taxonomies of attacks, tools and countermeasures. In *Proceedings of the International Workshop on Security in Parallel and Distributed Systems*, pages 543–550, 2004.

[57] Top 100 network security tools. WWW page, 2006. URL `http://www.sectools.org`. Last accessed: 23/11/2009.

[58] Nmap security scanner version 5.00. WWW page, 2009. URL `http://www.nmap.org`. Last accessed: 23/11/2009.

[59] The gnu netcat project. WWW page, January 2004. URL `http://netcat.sourceforge.net`. Last accessed: 23/11/2009.

[60] The metasploit project. WWW page, 2003. URL `http://www.metasploit.com`. Last accessed: 23/11/2009.

[61] Cisco network intrusion prevention system. WWW page, 2009. URL `http://www.cisco.com/cisco/web/UK/products/vpn.html`. Last accessed: 23/11/2009.

[62] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings IEEE Symposium on Security and Privacy*, pages 285 –283, 2002.

[63] Lambert Schaelicke, Kyle Wheeler, and Curt Freeland. Spanids: a scalable network intrusion detection loadbalancer. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 315–322, New York, NY, USA, 2005. ACM. ISBN 1-59593-019-1.

[64] Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Trans. Dependable Secur. Comput.*, 3(1):31, 2006. ISSN 1545-5971.

[65] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *RAID*, pages 107–126, 2007.

[66] D. Ficara, S. Giordano, F. Oppedisano, G. Procissi, and F. Vitucci. A cooperative pc/network-processor architecture for multi gigabit traffic analysis. In *Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International*, pages 123 –128, 2008.

[67] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Kon, and Ashley Thomas. A hardware platform for network intrusion detection and prevention. In *In Proceedings of the 3rd Workshop on Network Processors and Applications (NP3*, 2004.

[68] Christopher R. Clark and Craig D. Ulmer. Network intrusion detection system on fpgas with on-chip network interfaces. In *In proceedings of International Workshop on Applied Reconfigurable Computing (ARC)*, 02 2005.

[69] Byoungkoo Kim Seungyong Yoon and Jintae Oh. High-performance stateful intrusion detection system. In *Computational Intelligence and Security, 2006 International Conference*, pages 574–579, 11 2006.

[70] Vasiliadis Giorgos, Antonatos Spiros, Polychronakis Michalis, Markatos Evangelos, and Ioannidis Sotiris. Gnort: High performance network intrusion detection using graphics processors. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7.

[71] Domenico Ficara, Stefano Giordano, and Fabio Vitucci. Design and implementation of a multidimensional packet classifier on network processor. Technical report, Department of Information Engineering, University of Pisa, Italy, 2007. URL `http://wwwtlc.iet.unipi.it/research/classifier.pdf`. Last accessed: 21/06/2010.

[72] Mike Fisk and George Varghese. Fast content-based packet handling for intrusion detection. Technical report, Los Alamos National Laboratory, University of California San Diego, 2001. URL `http://woozle.org/~mfisk/papers/ucsd-tr-cs2001-0670.pdf`. Last accessed: 02 Dec.09.

[73] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. *DARPA Information Survivability Conference and Exposition*, 1:0367, 2001.

[74] Evangelos P. Markatos, Spyros Antonatos, Michalis Polychronakis, and Kostas G. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *In Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN*, pages 146–152. ACTA Press, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.5035`.

[75] Anagnostakis Antonatos Markatos, K. G. Anagnostakis, S. Antonatos, E. P. Markatos, and M. Polychronakis. E2xb: A domain-specific string matching algorithm for intrusion detection. In *In Proceedings of the 18th IFIP International*

Information Security Conference (SEC2003, 2003. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.1456`.

[76] S. Antonatos, M. Polychronakis, P. Akritidis, and K. G. Anagnostakis. Piranha: Fast and memory-efficient pattern matching for intrusion detection. In *In Proceedings 20th IFIP International Information Security Conference (SEC 2005*, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.74.1767`.

[77] Ioannis Sourdis, Vasilis Dimopoulos, Dionisios Pnevmatikatos, and Stamatis Vassiliadis. Packet pre-filtering for network intrusion detection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2006. ACM. ISBN 1-59593-580-0. URL `http://portal.acm.org/citation.cfm?id=1185372`.

[78] Michael Attig and John W. Lockwood. Sift: Snort intrusion filter for TCP. In *IEEE Symposium on High Performance Interconnects (HotI-13)*, Stanford, CA, aug 2005. URL `http://www.arl.wustl.edu/projects/fpx/references/SIFT_Lockwood_Attig-Hot_Interconnects_2005.pdf`.

[79] Haoyu Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: a reconfigurable hardware nids filter. *International Conference on Field Programmable Logic and Applications*, 0:493–498, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/FPL.2005.1515770. URL `http://www2.computer.org/portal/web/csdl/doi/10.1109/FPL.2005.1515770`.

[80] José M. González, Vern Paxson, and Nicholas Weaver. Shunting: a hardware/-software architecture for flexible, high-performance network intrusion prevention. In *ACM Conference on Computer and Communications Security*, pages 139–149, 2007.

[81] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24 (1):52–61, 01 2004.

[82] M Nourani. and P Katta. Bloom filter accelerator for string matching. In *Proceedings of 16th International Conference on Computer Communications and Networks, ICCCN 2007.*, pages 185–190, Honolulu, HI, 2007. IEEE. ISBN 978-1-4244-1251-8.

[83] A. Yang Chen, Kumar and Jun Xu. A new design of bloom filter for packet inspection speedup. In *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 1–5, Washington, DC, 2007. IEEE. ISBN 978-1-4244-1043-9.

[84] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection. *International Conference on Field Programmable Logic and Applications*, 0:644–647, 2005.

[85] Sherif Yusuf, Wayne Luk, M. K. N. Szeto, and William G. Osborne. Unite: Uniform hardware-based network intrusion detection engine. In *International workshop on Applied Reconfigurable Computing.*, 2006.

[86] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 258–267, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2230-0.

[87] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 404–413, London, UK, 2002. Springer-Verlag. ISBN 3-540-44108-5.

[88] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 112–122, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2270-X.

[89] Hong-Jip Jung, Z.K. Baker, and V.K. Prasanna. Performance of fpga implementation of bit-split architecture for intrusion detection systems. *Parallel and Distributed Processing Symposium, International*, 0:177, 2006.

[90] Young H. Cho and William H. Mangione-Smith. Deep network packet filter design for reconfigurable devices. *ACM Transaction Embedded Computing System*, 7(2): 1–26, 2008. ISSN 1539-9087.

[91] Rong-Tai Liu, Nen-Fu Huang, Chia-Nan Kao, Chih-Hao Chen, and Chi-Chieh Chou. A fast pattern-match engine for network processor-based network intrusion detection system. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 97, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2108-8.

[92] Luis Carlos Caruso, Guilherme Guindani, Hugo Schmitt, Ney Calazans, and Fernando Moraes. Spp-nids - a sea of processors platform for network intrusion detection systems. In *RSP '07: Proceedings of the 18th IEEE/IFIP International*

Workshop on Rapid System Prototyping, pages 27–33, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2834-1.

[93] Herbert Bos and Kaiming Huang. A network intrusion detection system on ixp1200 network processor. In *Technical Report, LIACS, Leiden University*, February 2004. URL `citeseer.ist.psu.edu/703003.html`.

[94] Michael Attig and John Lockwood. A framework for rule processing in reconfigurable network systems. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:225–234, 2005.

[95] R. Nigel Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[96] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *In proceedings of the IEEE Infocom, Hong Kong*, pages 333–340, March 2004.

[97] Marc Norton. Optimizing pattern matching for intrusion detection. Technical report, ., July 2004. URL `http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf`. Last accessed: 24 Nov. 2009.

[98] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994. URL `http://citeseer.ist.psu.edu/wu94fast.html`. Last accessed: 21/01/2010.

[99] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006. ACM.

[100] Zachary K. Baker and Viktor K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 193–202, New York, NY, USA, 2005. ACM. ISBN 1-59593-082-5.

[101] Giorgos Papadopoulos and Hardware Laboratory. Hashing + memory = low cost, exact pattern matching. In *In Proceedings of the 15th International Conference on Field Programmable Logic and Applications*, pages 39–44, 2005.

[102] B.C. Brodie, R.K. Cytron, and D.E. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 191–202, 2006.

[103] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-945-6.

[104] A. Kennedy, Xiaojun Wang, Zhen Liu, and Bin Liu. Ultra-high throughput string matching for deep packet inspection. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 8-12 2010.

[105] P. Piyachon and Yan Luo. Efficient memory utilization on network processors for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 71–80, 3-5 2006.

[106] Sarang Dharmapurikar and John W. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal*, 24(10):1781–1792, 10 2006.

[107] David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David Mcclung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *in Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, pages 12–26, 2000.

[108] Hudo Teufel III. Computer security threat monitoring and surveillance. Technical report, U.S. Department of Homeland Security, May 2008. URL `http://www.dhs.gov/xlibrary/assets/privacy/privacy_pia_einstein2.pdf`. Last accessed: 09 Mar. 2010.

[109] Metor graphics rc series. WWW page, 2010. URL `http://www.mentor.com/products/fpga/handel-c/rc-series-platforms/`. Last accessed: 23/12/2010.

[110] Patrick Schaumont and Ingrid Verbauwhede. Hardware/software codesign for stream ciphers. WWW page, 2007. URL `http://www.ecrypt.eu.org/stream/papersdir/2007/016.pdf`. Last accessed: 17/04/2010.

[111] P. Huerta, J. Castillo, J. I. Mártinez, and V. López. Multi microblaze system for parallel computing. In *ICC'05: Proceedings of the 9th International Conference on Circuits*, pages 1–6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS). ISBN 960-8457-29-7.

[112] Neil Desai. Increasing performance in high speed nids: A look at snort's internals. Technical report, ., 3 2002. URL `http://www.linuxsecurity.com/resource_files/intrusionvdetection/increasing_Performance_in_High_Speed_IDS.pdf`. Last accessed: 24 Nov. 2009.

[113] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[114] Jeff Yan and Pook Leong Cho. Enhancing collaborative spam detection with bloom filters. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 414–428, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2716-7.

[115] Guofei Gu, Monirul Sharif, Xinzhou Qin, David Dagon, Wenke Lee, and George Riley. Worm detection, early warning and response based on local victim information. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2252-1.

[116] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, New York, NY, USA, 1977. ACM.

[117] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *In Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[118] S. Brown, C. Deane, G. Ho, and P. Mucci. Papi:a portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[119] The shmoo group. WWW page, 2010. URL `http://cctf.shmoo.com/`. Last accessed: 23/12/2010.

APPENDIX A: Published Research

Adeel Hashmi and Dr. Andy Nisbet. Hardware/Software Co-design Platform for Network Intrusion Detection System. *In Proceedings of the Third International Conference on Internet Technologies and Applications*, Wales, September, 2009.

Adeel Hashmi and Dr. Andy Nisbet. Hardware/Software Co-design Platform for Network Intrusion Detection System. In Proceedings of the first MMU-RD-10 Science and Engineering Research and Development Conference, Manchester, December, 2010.